Nvim documentation

If you are new to Nvim, see nvim-intro. (To go to a help link, move the cursor to it and hit CTRL-], or right-click with mouse. Try it here: bars.)

About Nvim

Q_ct

- news News since the previous release
- nvim Getting started with Nvim
- tutor 30-minute interactive course for beginners
- vim-differences Nvim compared to Vim
- faq Frequently Asked Questions
- tips Various tips
- bugs Where to send bug reports
- support Supported platforms
- copying About copyrights

Usage

- helphelp Using the `:help` files
- intro Introduction to Vim
- quickref Overview of common commands
- index Index of all commands
- user-manual User manual: How to accomplish editing tasks.
- message.txt (Error) messages and explanations
- Kuwasha Helping poor children in Uganda

Basic editing

- starting Starting Vim, Vim command arguments, initialisation
- edit-files Editing and writing files
- motion.txt Commands for moving around
- scrolling Scrolling the text in the window
- insert.txt Insert and Replace mode
- change.txt Deleting and replacing text
- undo-redo Undo and Redo
- repeat.txt Repeating commands, Vim scripts and debugging
- visual-mode Using Visual mode (selecting text)
- various Various other commands
- crash-recovery Recovering from a crash

Advanced editing

- cmdline Command-line editing
- options Description of all options
- pattern-searches Vim regexp patterns and search commands
- key-mapping Key mapping (shortcuts), abbreviations
- tags Tags and special searches
- windows Commands for using windows and buffers
- tabpage Commands for using tabpages
- spell Spell checking
- diff Comparing files
- folding Hide (fold) ranges of lines
- terminal Embedded terminal emulator

API (extensibility/scripting/plugins)

- api Nvim API via RPC, Lua and Vimscript
- ui Nvim UI protocol
- lua-guide Nvim Lua guide
- lua Lua API
- luaref Lua reference manual
- luvref Luv (vim.uv) reference manual
- autocmd Event handlers
- job-control Spawn and control multiple processes
- channel Nvim asynchronous IO
- vimscript Vimscript reference
- vimscript-functions Vimscript functions
- remote-plugin Nvim remote plugins
- health Health checking

Programming language support

- diagnostic-api Diagnostic framework
- filetype Settings for specific types of files
- indent.txt automatic indenting for C and other languages
- lsp Language Server Protocol (LSP)

- quickfix Commands for a quick edit-compile-fix cycle
- syntax syntax highlighting
- treesitter Incremental syntax parsing
- vim.pack Plugin manager

UI

- tui Builtin UI
- gui External (graphical) UIs
- signs Signs displayed as window decorations (the "gutter")

Multilingual support

- digraph List of available digraphs
- mbyte.txt Multibyte text support
- mlang.txt Non-English language support
- rileft.txt Right-to-left editing mode
- l10n-arabic.txt Arabic language support and editing
- l10n-hebrew.txt Hebrew language support and editing
- l10n-russian.txt Russian language support and editing
- l10n-vietnamese.txt Vietnamese language support and editing

Interop

- provider Builtin remote plugin hosts
- if_perl Perl interface
- if_pyth Python interface
- if_ruby Ruby interface

Versions

- deprecated **Deprecated** features that will be removed
- vi-differences Differences between Vim and Vi

Developing nvim

- dev Development of Nvim
- dev-arch Internal architecture, modules, data structures
- dev-style Development style guidelines
- dev-test Writing and running tests
- dev-theme Design guidelines (colorschemes etc.)
- dev-tools Tools and techniques for developing Nvim
- dev-vimpatch Merging patches from Vim

Standard plugins

See standard-plugin-list.

Local additions

Bars example **bars**

Now that you've jumped here with CTRL-] or a double mouse click, you can use CTRL-T, CTRL-O, g<RightMouse>, or <C-RightMouse> to go back to where you were.

Note that tags are within | characters, but when highlighting is enabled these characters are hidden (for aesthetics).

You can use CTRL-] on any word (even if it is not within "|") and Nvim will try to find help for it. Especially for options in single quotes, e.g. 'hlsearch'.

VIM USER MANUAL by Bram Moolenaar

Table Of Contents usr

Overview

Getting Started

usr_01.txt About the manuals
usr_02.txt The first steps in Vim
usr_03.txt Moving around
usr_04.txt Making small changes
usr_05.txt Set your settings
usr_06.txt Using syntax highlighting
usr_07.txt Editing more than one file
usr_08.txt Splitting windows
usr_09.txt Using the GUI
usr_10.txt Making big changes
usr_11.txt Recovering from a crash
usr_12.txt Clever tricks

Editing Effectively

usr_20.txt Typing command-line commands quickly
usr_21.txt Go away and come back
usr_22.txt Finding the file to edit
usr_23.txt Editing other files
usr_24.txt Inserting quickly
usr_25.txt Editing formatted text
usr_26.txt Repeating
usr_27.txt Search commands and patterns
usr_28.txt Folding
usr_29.txt Moving through programs
usr_30.txt Editing programs
usr_31.txt Exploiting the GUI
usr_32.txt The undo tree

Tuning Vim

usr_40.txt Make new commands
usr_41.txt Write a Vim script
usr_42.txt Add new menus
usr_43.txt Using filetypes
usr_44.txt Your own syntax highlighted
usr_45.txt Select your language (locale)

Reference manual

reference_toc More detailed information for all commands

The user manual is online:
https://neovim.io/doc/user

Getting Started

Read this from start to end to learn the essential commands.

usr_01.txt About the manuals
    01.1 Two manuals
    01.2 Vim installed
    01.3 Using the Vim tutor
    01.4 Copyright

usr_02.txt The first steps in Vim
    02.1 Running Vim for the First Time
    02.2 Inserting text
    02.3 Moving around
    02.4 Deleting characters
    02.5 Undo and Redo
    02.6 Other editing commands
    02.7 Getting out
    02.8 Finding help

usr_03.txt Moving around
    03.1 Word movement
    03.2 Moving to the start or end of a line
    03.3 Moving to a character
    03.4 Matching a paren
    03.5 Moving to a specific line
    03.6 Telling where you are
    03.7 Scrolling around
    03.8 Simple searches
    03.9 Simple search patterns
    03.10 Using marks

usr_04.txt Making small changes
    04.1 Operators and motions
    04.2 Changing text
    04.3 Repeating a change
    04.4 Visual mode
    04.5 Moving text
    04.6 Copying text
    04.7 Using the clipboard
    04.8 Text objects
    04.9 Replace mode
    04.10 Conclusion

usr_05.txt Set your settings
    05.1   The vimrc file
    05.2   The example vimrc file explained
    05.3   Simple mappings
    05.4   Adding a package
    05.5   Adding a plugin
    05.6   Adding a help file
    05.7   The option window
    05.8   Often used options

usr_06.txt Using syntax highlighting
    06.1   Switching it on
    06.2   No or wrong colors?
    06.3   Different colors
    06.4   With colors or without colors
    06.5   Further reading

usr_07.txt Editing more than one file
    07.1   Edit another file
    07.2   A list of files
    07.3   Jumping from file to file
    07.4   Backup files
    07.5   Copy text between files
    07.6   Viewing a file
    07.7   Changing the file name

usr_08.txt Splitting windows
    08.1   Split a window
    08.2   Split a window on another file
    08.3   Window size
    08.4   Vertical splits
    08.5   Moving windows
    08.6   Commands for all windows
    08.7   Viewing differences with diff mode
    08.8   Various

usr_09.txt Using the GUI
    09.1   Parts of the GUI
    09.2   Using the mouse
    09.3   The clipboard
    09.4   Select mode

usr_10.txt Making big changes
    10.1   Record and playback commands
    10.2   Substitution
    10.3   Command ranges
    10.4   The global command
    10.5   Visual block mode
    10.6   Reading and writing part of a file
    10.7   Formatting text
    10.8   Changing case
    10.9   Using an external program

usr_11.txt Recovering from a crash
    11.1   Basic recovery
    11.2   Where is the swap file?
    11.3   Crashed or not?
    11.4   Further reading

usr_12.txt Clever tricks
    12.1   Replace a word
    12.2   Change "Last, First" to "First Last"
    12.3   Sort a list
    12.4   Reverse line order
    12.5   Count words
    12.6   Find a man page
    12.7   Trim blanks
    12.8   Find where a word is used

Editing Effectively

Subjects that can be read independently.

usr_20.txt Typing command-line commands quickly
    20.1   Command line editing
    20.2   Command line abbreviations
    20.3   Command line completion
    20.4   Command line history
    20.5   Command line window

usr_21.txt Go away and come back
    21.1   Suspend and resume
    21.2   Executing shell commands
    21.3   Remembering information; ShaDa
    21.4   Sessions
    21.5   Views
    21.6   Modelines

usr_22.txt Finding the file to edit
    22.1   The file explorer
    22.2   The current directory
    22.3   Finding a file
    22.4   The buffer list

usr_23.txt Editing other files
    23.1   DOS, Mac and Unix files

23.2 Files on the internet
23.3 Binary files
23.4 Compressed files

usr_24.txt Inserting quickly
24.1 Making corrections
24.2 Showing matches
24.3 Completion
24.4 Repeating an insert
24.5 Copying from another line
24.6 Inserting a register
24.7 Abbreviations
24.8 Entering special characters
24.9 Digraphs
24.10 Normal mode commands

usr_25.txt Editing formatted text
25.1 Breaking lines
25.2 Aligning text
25.3 Indents and tabs
25.4 Dealing with long lines
25.5 Editing tables

usr_26.txt Repeating
26.1 Repeating with Visual mode
26.2 Add and subtract
26.3 Making a change in many files
26.4 Using Vim from a shell script

usr_27.txt Search commands and patterns
27.1 Ignoring case
27.2 Wrapping around the file end
27.3 Offsets
27.4 Matching multiple times
27.5 Alternatives
27.6 Character ranges
27.7 Character classes
27.8 Matching a line break
27.9 Examples

usr_28.txt Folding
28.1 What is folding?
28.2 Manual folding
28.3 Working with folds
28.4 Saving and restoring folds
28.5 Folding by indent
28.6 Folding with markers
28.7 Folding by syntax
28.8 Folding by expression
28.9 Folding unchanged lines
28.10 Which fold method to use?

usr_29.txt Moving through programs
29.1 Using tags
29.2 The preview window
29.3 Moving through a program
29.4 Finding global identifiers
29.5 Finding local identifiers

usr_30.txt Editing programs
30.1 Compiling
30.2 Indenting C files
30.3 Automatic indenting
30.4 Other indenting
30.5 Tabs and spaces
30.6 Formatting comments

usr_31.txt Exploiting the GUI
31.1 The file browser
31.2 Confirmation
31.3 Menu shortcuts
31.4 Vim window position and size
31.5 Various

usr_32.txt The undo tree
32.1 Undo up to a file write
32.2 Numbering changes
32.3 Jumping around the tree
32.4 Time travelling

Tuning Vim

Make Vim work as you like it.

usr_40.txt Make new commands
40.1 Key mapping
40.2 Defining command-line commands
40.3 Autocommands

usr_41.txt Write a Vim script
41.1 Introduction
41.2 Variables
41.3 Expressions
41.4 Conditionals
41.5 Executing an expression
41.6 Using functions

41.7 Defining a function
41.8 Lists and Dictionaries
41.9 Exceptions
41.10 Various remarks
41.11 Writing a plugin
41.12 Writing a filetype plugin
41.13 Writing a compiler plugin
41.14 Writing a plugin that loads quickly
41.15 Writing library scripts
41.16 Distributing Vim scripts

usr_42.txt Add new menus
42.1 Introduction
42.2 Menu commands
42.3 Various
42.4 Toolbar and popup menus

usr_43.txt Using filetypes
43.1 Plugins for a filetype
43.2 Adding a filetype

usr_44.txt Your own syntax highlighted
44.1 Basic syntax commands
44.2 Keywords
44.3 Matches
44.4 Regions
44.5 Nested items
44.6 Following groups
44.7 Other arguments
44.8 Clusters
44.9 Including another syntax file
44.10 Synchronizing
44.11 Installing a syntax file
44.12 Portable syntax file layout

usr_45.txt Select your language (locale)
45.1 Language for Messages
45.2 Language for Menus
45.3 Using another encoding
45.4 Editing files with a different encoding
45.5 Entering language text

Copyright: see manual-copyright

VIM USER MANUAL by Bram Moolenaar

About the manuals

This chapter introduces the manuals available with Vim. Read this to know the conditions under which the commands are explained.

01.1 Two manuals
01.2 Vim installed
01.3 Using the Vim tutor
01.4 Copyright

Next chapter: usr_02.txt The first steps in Vim
Table of contents: usr_toc.txt

Two manuals

The Vim documentation consists of two parts:

1. The User manual

   Task oriented explanations, from simple to complex. Reads from start to end like a book.
2. The Reference manual

   Precise description of how everything in Vim works.

The notation used in these manuals is explained here: notation

JUMPING AROUND

The text contains hyperlinks between the two parts, allowing you to quickly jump between the description of an editing task and a precise explanation of the commands and options used for it. Use these two commands:

```
Press CTRL-] to jump to a subject under the cursor.
Press CTRL-O to jump back (repeat to go further back).
```

Many links are in vertical bars, like this: bars. The bars themselves may be hidden or invisible; see below. An option name, like 'number', a command in double quotes like ":write" and any other word can also be used as a link. Try it out: Move the cursor to CTRL-] and press CTRL-] on it.

Other subjects can be found with the ":help" command; see help.txt.

The bars and stars are usually hidden with the conceal feature. They also use hl-Ignore, using the same color for the text as the background. You can make them visible with:

```
:set conceallevel=0
:hi link HelpBar Normal
:hi link HelpStar Normal
```

01.2 Vim installed setup-vimrc_example

To create an empty vimrc:

```
:call mkdir(stdpath('config'),'p')
:exe 'edit' stdpath('config').'/init.vim'
:write
```

For more info see vimrc.

01.3 Using the Vim tutor tutor vimtutor

Instead of reading the text (boring!) you can use :Tutor to learn your first Vim commands. This is a 30-minute tutorial that teaches the most basic Vim functionality hands-on.

To start the tutorial, execute

```
:Tutor
```

from within nvim. The tutorial will lead you from that point. Have fun!

01.4 Copyright manual-copyright

The Vim user manual and reference manual are Copyright (c) 1988 by Bram Moolenaar. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later. The latest version is presently available at:
https://opencontent.org/openpub/

People who contribute to the manuals must agree with the above copyright notice.

frombook

Parts of the user manual come from the book "Vi IMproved - Vim" by Steve Oualline (published by New Riders Publishing, ISBN: 0735710015). The Open Publication License applies to this book. Only selected parts are included

and these have been modified (e.g., by removing the pictures, updating the text for Vim 6.0 and later, fixing mistakes). The omission of the frombook tag does not mean that the text does not come from the book.

Many thanks to Steve Oualline and New Riders for creating this book and publishing it under the OPL! It has been a great help while writing the user manual. Not only by providing literal text, but also by setting the tone and style.

If you make money through selling the manuals, you are strongly encouraged to donate part of the profit to help AIDS victims in Uganda. See Kuwasha.

Next chapter: usr_02.txt The first steps in Vim

Copyright: see manual-copyright

VIM USER MANUAL by Bram Moolenaar

The first steps in Vim

This chapter provides just enough information to edit a file with Vim. Not well or fast, but you can edit. Take some time to practice with these commands, they form the base for what follows.

02.1 Running Vim for the First Time
02.2 Inserting text
02.3 Moving around
02.4 Deleting characters
02.5 Undo and Redo
02.6 Other editing commands
02.7 Getting out
02.8 Finding help

Next chapter: usr_03.txt Moving around
Previous chapter: usr_01.txt About the manuals
Table of contents: usr_toc.txt

Running Vim for the First Time

To start Nvim, enter this command:

```
nvim file.txt
```

On Unix you can type this at any command prompt. If you are running Microsoft Windows, open a Command Prompt and enter the command. In either case, Vim starts editing a file called file.txt. Because this is a new file, you get a blank window. This is what your screen will look like:

```
+----------------------------------------+
|#                                       |
|~                                       |
|~                                       |
|~                                       |
|~                                       |
|"file.txt" [New]                        |
+----------------------------------------+
        ('#' is the cursor position.)
```

The tilde (~) lines indicate lines not in the file. In other words, when Vim runs out of file to display, it displays tilde lines. At the bottom of the screen, a message line indicates the file is named file.txt and shows that you are creating a new file. The message information is temporary and other information overwrites it.

02.2 Inserting text

The Vim editor is a modal editor. That means that the editor behaves differently, depending on which mode you are in. The two basic modes are called Normal mode and Insert mode. In Normal mode the characters you type are commands. In Insert mode the characters are inserted as text.

Since you have just started Vim it will be in Normal mode. To start Insert mode you type the "i" command (i for Insert). Then you can enter the text. It will be inserted into the file. Do not worry if you make mistakes; you can correct them later. To enter the following programmer's limerick, this is what you type:

```
iA very intelligent turtle
Found programming Unix a hurdle
```

After typing "turtle" you press the <Enter> key to start a new line. Finally you press the <Esc> key to stop Insert mode and go back to Normal mode. You now have two lines of text in your Vim window:

```
+----------------------------------------+
|A very intelligent turtle               |
|Found programming Unix a hurdle         |
|~                                       |
|~                                       |
|                                        |
+----------------------------------------+
```

WHAT IS THE MODE?

To be able to see what mode you are in, type this command:

```
:set showmode
```

You will notice that when typing the colon Vim moves the cursor to the last line of the window. That's where you type colon commands (commands that start with a colon). Finish this command by pressing the <Enter> key (all commands that start with a colon are finished this way).

Now, if you type the "i" command Vim will display --INSERT-- at the bottom of the window. This indicates you are in Insert mode.

```
+--------------------------------------+
|A very intelligent turtle             |
|Found programming Unix a hurdle       |
|~                                     |
|~                                     |
|-- INSERT --                          |
+--------------------------------------+
```

If you press <Esc> to go back to Normal mode the last line will be made blank.

GETTING OUT OF TROUBLE

One of the problems for Vim novices is mode confusion, which is caused by forgetting which mode you are in or by accidentally typing a command that switches modes. To get back to Normal mode, no matter what mode you are in, press the <Esc> key. Sometimes you have to press it twice. If Vim beeps back at you, you already are in Normal mode.

02.3 Moving around

After you return to Normal mode, you can move around by using these keys:

hjkl

```
        h   left

        j   down
        k   up
        l   right
```

At first, it may appear that these commands were chosen at random. After all, who ever heard of using l for right? But actually, there is a very good reason for these choices: Moving the cursor is the most common thing you do in an editor, and these keys are on the home row of your right hand. In other words, these commands are placed where you can type them the fastest (especially when you type with ten fingers).

> **Note:**
> You can also move the cursor by using the arrow keys. If you do, however, you greatly slow down your editing because to press the arrow keys, you must move your hand from the text keys to the arrow keys. Considering that you might be doing it hundreds of times an hour, this can take a significant amount of time.
> Also, there are keyboards which do not have arrow keys, or which locate them in unusual places; therefore, knowing the use of the hjkl keys helps in those situations.

One way to remember these commands is that h is on the left, l is on the right and j points down. In a picture:

```
    k
h     l
  j
```

The best way to learn these commands is by using them. Use the "i" command to insert some more lines of text. Then use the hjkl keys to move around and insert a word somewhere. Don't forget to press <Esc> to go back to Normal mode. :Tutor is also a nice way to learn by doing.

For Japanese users, Hiroshi Iwatani suggested using this:

```
                       Komsomolsk
                           ^
                           |
          Huan Ho      <--- --->  Los Angeles
        (Yellow river)     |
                           v
                         Java (the island, not the programming language)
```

02.4 Deleting characters

To delete a character, move the cursor over it and type "x". (This is a throwback to the old days of the typewriter, when you deleted things by typing xxxx over them.) Move the cursor to the beginning of the first line, for example, and type xxxxxxx (seven x's) to delete "A very ". The result should look like this:

```
+--------------------------------------+
|intelligent turtle                    |
|Found programming Unix a hurdle       |
|~                                     |
|~                                     |
|                                      |
+--------------------------------------+
```

Now you can insert new text, for example by typing:

```
iA young <Esc>
```

This begins an insert (the i), inserts the words "A young", and then exits insert mode (the final <Esc>). The result:

```
+--------------------------------------+
|A young intelligent turtle            |
|Found programming Unix a hurdle       |
|~                                     |
|~                                     |
|                                      |
+--------------------------------------+
```

DELETING A LINE

To delete a whole line use the "dd" command. The following line will then move up to fill the gap:

```
+--------------------------------------+
|Found programming Unix a hurdle       |
|~                                     |
|~                                     |
|~                                     |
|                                      |
+--------------------------------------+
```

DELETING A LINE BREAK

In Vim you can join two lines together, which means that the line break between them is deleted. The "J" command does this.
Take these two lines:

A young intelligent
turtle

Move the cursor to the first line and press "J":

A young intelligent turtle

02.5 Undo and Redo

Suppose you delete too much. Well, you can type it in again, but an easier way exists. The "u" command undoes the last edit. Take a look at this in action: After using "dd" to delete the first line, "u" brings it back.
Another one: Move the cursor to the A in the first line:

A young intelligent turtle

Now type xxxxxxx to delete "A young". The result is as follows:

intelligent turtle

Type "u" to undo the last delete. That delete removed the g, so the undo restores the character.

g intelligent turtle

The next "u" command restores the next-to-last character deleted:

ng intelligent turtle

The next "u" command gives you the u, and so on:

ung intelligent turtle
oung intelligent turtle
young intelligent turtle
young intelligent turtle
A young intelligent turtle

REDO

If you undo too many times, you can press CTRL-R (redo) to reverse the preceding command. In other words, it undoes the undo. To see this in action, press CTRL-R twice. The character A and the space after it disappear:

young intelligent turtle

There's a special version of the undo command, the "U" (undo line) command. The undo line command undoes all the changes made on the last line that was edited. Typing this command twice cancels the preceding "U".

A very intelligent turtle
xxxx Delete very

A intelligent turtle
xxxxxx Delete turtle

A intelligent
Restore line with "U"

A very intelligent turtle
Undo "U" with "u"

A intelligent

The "U" command is a change by itself, which the "u" command undoes and CTRL-R redoes. This might be a bit confusing. Don't worry, with "u" and CTRL-R you can go to any of the situations you had. More about that in section 32.2.

02.6 Other editing commands

Vim has a large number of commands to change the text. See O_in and below. Here are a few often used ones.

APPENDING

The "i" command inserts a character before the character under the cursor. That works fine; but what happens if you want to add stuff to the end of the line? For that you need to insert text after the cursor. This is done with the "a" (append) command.
For example, to change the line

and that's not saying much for the turtle.
to
and that's not saying much for the turtle!!!

move the cursor over to the dot at the end of the line. Then type "x" to delete the period. The cursor is now positioned at the end of the line on the e in turtle. Now type

```
a!!!<Esc>
```

to append three exclamation points after the e in turtle:

and that's not saying much for the turtle!!!

OPENING UP A NEW LINE

The "o" command creates a new, empty line below the cursor and puts Vim in Insert mode. Then you can type the text for the new line.
Suppose the cursor is somewhere in the first of these two lines:

A very intelligent turtle
Found programming Unix a hurdle

If you now use the "o" command and type new text:

```
oThat liked using Vim<Esc>
```

The result is:

A very intelligent turtle
That liked using Vim
Found programming Unix a hurdle

The "O" command (uppercase) opens a line above the cursor.

USING A COUNT

Suppose you want to move up nine lines. You can type "kkkkkkkkk" or you can enter the command "9k". In fact, you can precede many commands with a number. Earlier in this chapter, for instance, you added three exclamation points to the end of a line by typing "a!!!<Esc>". Another way to do this is to use the command "3a!<Esc>". The count of 3 tells the command that follows to triple its effect. Similarly, to delete three characters, use the command "3x". The count always comes before the command it applies to.

02.7 Getting out

To exit, use the "ZZ" command. This command writes the file and exits.

> **Note:**
> Unlike many other editors, Vim does not automatically make a backup file. If you type "ZZ", your changes are committed and there's no turning back. You can configure the Vim editor to produce backup files; see 07.4.

DISCARDING CHANGES

Sometimes you will make a sequence of changes and suddenly realize you were better off before you started. Not to worry; Vim has a quit-and-throw-things-away command. It is:

```
:q!
```

Don't forget to press <Enter> to finish the command.

For those of you interested in the details, the three parts of this command are the colon (:), which enters Command-line mode; the q command, which tells the editor to quit; and the override command modifier (!).
The override command modifier is needed because Vim is reluctant to throw away changes. If you were to just type ":q", Vim would display an error message and refuse to exit:

E37: No write since last change (use ! to override)

By specifying the override, you are in effect telling Vim, "I know that what I'm doing looks stupid, but I really want to do this."

If you want to continue editing with Vim: The ":e!" command reloads the original version of the file.

02.8 Finding help

Everything you always wanted to know can be found in the Vim help files. Don't be afraid to ask!

If you know what you are looking for, it is usually easier to search for it using the help system, instead of using Google. Because the subjects follow a certain style guide.

Also the help has the advantage of belonging to your particular Vim version. You won't see help for commands added later. These would not work for you.

To get generic help use this command:

```
:help
```

You could also use the first function key <F1>. If your keyboard has a <Help> key it might work as well.
   If you don't supply a subject, ":help" displays the general help window. The creators of Vim did something very clever (or very lazy) with the help system: They made the help window a normal editing window. You can use all the normal Vim commands to move through the help information. Therefore h, j, k, and l move left, down, up and right.
   To get out of the help window, use the same command you use to get out of the editor: "ZZ". This will only close the help window, not exit Vim.

As you read the help text, you will notice some text enclosed in vertical bars (for example, help). This indicates a hyperlink. If you position the cursor anywhere between the bars and press CTRL-] (jump to tag), the help system takes you to the indicated subject. (For reasons not discussed here, the Vim terminology for a hyperlink is tag. So CTRL-] jumps to the location of the tag given by the word under the cursor.)
   After a few jumps, you might want to go back. CTRL-T (pop tag) takes you back to the preceding position. CTRL-O (jump to older position) also works nicely here.
   At the top of the help screen, there is the notation "*help.txt*". This name between "*" characters is used by the help system to define a tag (hyperlink destination).
   See 29.1 for details about using tags.

To get help on a given subject, use the following command:

```
:help {subject}
```

To get help on the "x" command, for example, enter the following:

```
:help x
```

To find out how to delete text, use this command:

```
:help deleting
```

To get a complete index of all Vim commands, use the following command:

```
:help index
```

When you need to get help for a control character command (for example, CTRL-A), you need to spell it with the prefix "CTRL-".

```
:help CTRL-A
```

The Vim editor has many different modes. By default, the help system displays the normal-mode commands. For example, the following command displays help for the normal-mode CTRL-H command:

```
:help CTRL-H
```

To identify other modes, use a mode prefix. If you want the help for the insert-mode version of a command, use "i_". For CTRL-H this gives you the following command:

```
:help i_CTRL-H
```

When you start the Vim editor, you can use several command-line arguments. These all begin with a dash (-). To find what the -t argument does, for example, use the command:

```
:help -t
```

The Vim editor has a number of options that enable you to configure and customize the editor. If you want help for an option, you need to enclose it in single quotation marks. To find out what the 'number' option does, for example, use the following command:

```
:help 'number'
```

The table with all mode prefixes can be found below: help-summary.

Special keys are enclosed in angle brackets. To find help on the up-arrow key
in Insert mode, for instance, use this command:

```
:help i_<Up>
```

If you see an error message that you don't understand, for example:

E37: No write since last change (use ! to override)

You can use the error ID at the start to find help about it:

```
:help E37
```

Summary: help-summary

1) Use Ctrl-D after typing a topic and let Vim show all available topics.
   Or press Tab to complete:

```
:help some<Tab>
```

   More information on how to use the help:

```
:help helphelp
```

2) Follow the links in bars to related help. You can go from the detailed
   help to the user documentation, which describes certain commands more from
   a user perspective and less detailed. E.g. after:

```
:help pattern.txt
```

   You can see the user guide topics 03.9 and usr_27.txt in the
   introduction.

3) Options are enclosed in single apostrophes. To go to the help topic for
   the list option:

```
:help 'list'
```

   If you only know you are looking for a certain option, you can also do:

```
:help options.txt
```

   to open the help page which describes all option handling and then search
   using regular expressions, e.g. textwidth.
   Certain options have their own namespace, e.g.:

```
:help cpo-<letter>
```

   for the corresponding flag of the 'cpoptions' settings, substitute <letter>
   by a specific flag, e.g.:

```
:help cpo-;
```

   And for the 'guioptions' flags:

```
:help go-<letter>
```

4) Normal mode commands do not have a prefix. To go to the help page for the
   "gt" command:

```
:help gt
```

5) Insert mode commands start with i_. Help for deleting a word:

```
:help i_CTRL-W
```

6) Visual mode commands start with v_. Help for jumping to the other side of
   the Visual area:

```
:help v_o
```

7) Command line editing and arguments start with c_. Help for using the
   command argument %:

```
:help c_%
```

8) Ex-commands always start with ":", so to go to the ":s" command help:

```
:help :s
```

9) Commands specifically for debugging start with ">". To go to the help
   for the "cont" debug command:

```
:help >cont
```

10) Key combinations. They usually start with a single letter indicating
    the mode for which they can be used. E.g.:

```
:help i_CTRL-X
```

    takes you to the family of CTRL-X commands for insert mode which can be
    used to auto-complete different things. Note, that certain keys will
    always be written the same, e.g. Control will always be CTRL.
    For normal mode commands there is no prefix and the topic is available at
    :h CTRL-<Letter>. E.g.

```
:help CTRL-W
```

    In contrast

```
:help c_CTRL-R
```

    will describe what the CTRL-R does when entering commands in the Command
    line and

```
:help v_CTRL-A
```

    talks about incrementing numbers in visual mode and

```
:help g_CTRL-A
```

    talks about the "g<C-A>" command (e.g. you have to press "g" then
    <CTRL-A>). Here the "g" stands for the normal command "g" which always
    expects a second key before doing something similar to the commands
    starting with "z".

11) Regexp items always start with /. So to get help for the "\+" quantifier
    in Vim regexes:

```
:help /\+
```

    If you need to know everything about regular expressions, start reading
    at:

```
:help pattern.txt
```

12) Registers always start with "quote". To find out about the special ":"
    register:

```
:help quote:
```

13) Vim script is available at

```
:help vimeval.txt
```

    Certain aspects of the language are available at :h expr-X where "X" is a
    single letter. E.g.

```
:help expr-!
```

    will take you to the topic describing the "!" (Not) operator for Vim
    Script.
    Also important is

```
:help function-list
```

    to find a short description of all functions available. Help topics for
    Vim script functions always include the "()", so:

```
:help append()
```

    talks about the append Vim script function rather than how to append text
    in the current buffer.

14) Mappings are talked about in the help page :h map.txt. Use

```
:help mapmode-i
```

    to find out about the :imap command. Also use :map-topic
    to find out about certain subtopics particular for mappings. e.g:

```
:help :map-local
```

    for buffer-local mappings or

```
:help map-bar
```

for how the '|' is handled in mappings.

15) Command definitions are talked about :h command-topic, so use

```
:help command-bang
```

to find out about the '!' argument for custom commands.

16) Window management commands always start with CTRL-W, so you find the
corresponding help at :h CTRL-W_letter. E.g.

```
:help CTRL-W_p
```

for moving the previous accessed window. You can also access

```
:help windows.txt
```

and read your way through if you are looking for window handling
commands.

17) Use :helpgrep to search in all help pages (and also of any installed
plugins). See :helpgrep for how to use it.
To search for a topic:

```
:helpgrep topic
```

This takes you to the first match. To go to the next one:

```
:cnext
```

All matches are available in the quickfix window which can be opened
with:

```
:copen
```

Move around to the match you like and press Enter to jump to that help.

18) The user manual. This describes help topics for beginners in a rather
friendly way. Start at usr_toc.txt to find the table of content (as you
might have guessed):

```
:help usr_toc.txt
```

Skim over the contents to find interesting topics. The "Digraphs" and
"Entering special characters" items are in chapter 24, so to go to that
particular help page:

```
:help usr_24.txt
```

Also if you want to access a certain chapter in the help, the chapter
number can be accessed directly like this:

```
:help 10.1
```

which goes to chapter 10.1 in usr_10.txt and talks about recording
macros.

19) Highlighting groups. Always start with hl-groupname. E.g.

```
:help hl-WarningMsg
```

talks about the WarningMsg highlighting group.

20) Syntax highlighting is namespaced to :syn-topic. E.g.

```
:help :syn-conceal
```

talks about the conceal argument for the ":syn" command.

21) Quickfix commands usually start with :c while location list commands
usually start with :l

22) Autocommand events can be found by their name:

```
:help BufWinLeave
```

To see all possible events:

```
:help events
```

23) Command-line switches always start with "-". So for the help of the -f
command switch of Vim use:

```
:help -f
```

24) Lua language and Nvim's Lua standard library are available at

```
:help lua.txt
```

Guide to using Lua in Nvim is available at

```
:help lua-guide.txt
```

Lua 5.1 reference manual is available at

```
:help luaref.txt
```

25) Documentation for included filetype specific functionality is usually
available in the form ft-<filetype>-<functionality>. So

```
:help ft-c-syntax
```

talks about the C syntax file and the option it provides. Sometimes,
additional sections for omni completion

```
:help ft-php-omni
```

or filetype plugins

```
:help ft-tex-plugin
```

are available.

26) Error and Warning codes can be looked up directly in the help. So

```
:help E297
```

takes you exactly to the description of the swap error message and

```
:help W10
```

talks about the warning "Changing a readonly file".
Sometimes, however, those error codes are not described, but rather are
listed at the Vim command that usually causes this. So:

```
:help E128
```

takes you to the :function command

27) Documentation for packages distributed with Vim have the form
package-<name>. So

```
:help package-termdebug
```

will bring you to the help section for the included termdebug plugin and
how to enable it.

Next chapter: usr_03.txt Moving around

Copyright: see manual-copyright

VIM USER MANUAL by Bram Moolenaar

Moving around

Before you can insert or delete text the cursor has to be moved to the right place. Vim has a large number of commands to position the cursor. This chapter shows you how to use the most important ones. You can find a list of these commands below Q_lr.

03.1 Word movement
03.2 Moving to the start or end of a line
03.3 Moving to a character
03.4 Matching a parenthesis
03.5 Moving to a specific line
03.6 Telling where you are
03.7 Scrolling around
03.8 Simple searches
03.9 Simple search patterns
03.10 Using marks

Next chapter: usr_04.txt Making small changes
Previous chapter: usr_02.txt The first steps in Vim
Table of contents: usr_toc.txt

Word movement

To move the cursor forward one word, use the "w" command. Like most Vim commands, you can use a numeric prefix to move past multiple words. For example, "3w" moves three words. This figure shows how it works (starting at the position marked with "x"):

```
This is a line with example text
          x-->-->->----------------->
           w  w  w    3w
```

Notice that "w" moves to the start of the next word if it already is at the start of a word.
   The "b" command moves backward to the start of the previous word:

```
This is a line with example text
        <----<--<-<---------<--x
           b   b b    2b      b
```

There is also the "e" command that moves to the next end of a word and "ge", which moves to the previous end of a word:

```
This is a line with example text
           <----<----x---->------------>
           2ge   ge     e       2e
```

If you are at the last word of a line, the "w" command will take you to the first word in the next line. Thus you can use this to move through a paragraph, much faster than using "l". "b" does the same in the other direction.

A word ends at a non-word character, such as a ".", "-" or ")". To change what Vim considers to be a word, see the 'iskeyword' option. If you try this out in the help directly, 'iskeyword' needs to be reset for the examples to work:

```
:set iskeyword&
```

It is also possible to move by white-space separated WORDs. This is not a word in the normal sense, that's why the uppercase is used. The commands for moving by WORDs are also uppercase, as this figure shows:

```
               ge      b          w                            e
               <-     <-         --->                         --->
This is-a line, with special/separated/words (and some more).
           <----- <-----         -------------------->        ----->
             gE      B                  W                      E
```

With this mix of lowercase and uppercase commands, you can quickly move forward and backward through a paragraph.

03.2 Moving to the start or end of a line

The "$" command moves the cursor to the end of a line. If your keyboard has an <End> key it will do the same thing.

The "^" command moves to the first non-blank character of the line. The "0" command (zero) moves to the very first character of the line, and the <Home> key does the same thing. In a picture ("." indicates a space):

```
                 ^
             <-----------x
.....This is a line with example text
        <---------------x   x-------------->
                  0                 $
```

(the "....." indicates blanks here)

   The "$" command takes a count, like most movement commands. But moving to the end of the line several times doesn't make sense. Therefore it causes the editor to move to the end of another line. For example, "1$" moves you to the end of the first line (the one you're on), "2$" to the end of the next line, and so on.

The "0" command doesn't take a count argument, because the "0" would be part of the count. Unexpectedly, using a count with "^" doesn't have any effect.

03.3 Moving to a character

One of the most useful movement commands is the single-character search command. The command "fx" searches forward in the line for the single character x. Hint: "f" stands for "Find".

For example, you are at the beginning of the following line. Suppose you want to go to the h of human. Just execute the command "fh" and the cursor will be positioned over the h:

```
To err is human.  To really foul up you need a computer.
        ---------->--------------->
            fh          fy
```

This also shows that the command "fy" moves to the end of the word really.

You can specify a count; therefore, you can go to the "l" of "foul" with "3fl":

```
To err is human.  To really foul up you need a computer.
                  --------------------->
                           3fl
```

The "F" command searches to the left:

```
To err is human.  To really foul up you need a computer.
                 <---------------------
                           Fh
```

The "tx" command works like the "fx" command, except it stops one character before the searched character. Hint: "t" stands for "To". The backward version of this command is "Tx".

```
To err is human.  To really foul up you need a computer.
                   <------------  ------------->
                        Th             tn
```

These four commands can be repeated with ";". "," repeats in the other direction. The cursor is never moved to another line. Not even when the sentence continues.

Sometimes you will start a search, only to realize that you have typed the wrong command. You type "f" to search backward, for example, only to realize that you really meant "F". To abort a search, press <Esc>. So "f<Esc>" is an aborted forward search and doesn't do anything. **Note:** <Esc> cancels most operations, not just searches.

03.4 Matching a parenthesis

When writing a program you often end up with nested () constructs. Then the "%" command is very handy: It moves to the matching paren. If the cursor is on a "(" it will move to the matching ")". If it's on a ")" it will move to the matching "(".

```
                           %
                         <----->
if (a == (b * c) / d)
                   <---------------->
                           %
```

This also works for [] and {} pairs. (This can be defined with the 'matchpairs' option.)

When the cursor is not on a useful character, "%" will search forward to find one. Thus if the cursor is at the start of the line of the previous example, "%" will search forward and find the first "(". Then it moves to its match:

```
if (a == (b * c) / d)
                ---+---------------->
                   %
```

Other ways to move around code can be found in usr_29.txt.

03.5 Moving to a specific line

If you are a C or C++ programmer, you are familiar with error messages such as the following:

```
prog.c:33: j   undeclared (first use in this function)
```

This tells you that you might want to fix something on line 33. So how do you find line 33? One way is to do "9999k" to go to the top of the file and "32j" to go down thirty-two lines. It is not a good way, but it works. A much better way of doing things is to use the "G" command. With a count, this command positions you at the given line number. For example, "33G" puts you on line 33. (For a better way of going through a compiler's error list, see usr_30.txt, for information on the :make command.)

With no argument, "G" positions you at the end of the file. A quick way to go to the start of a file use "gg". "1G" will do the same, but is a tiny bit more typing.

```
            |   first line of a file   ^
            |   text text text text    |
            |   text text text text    |  gg
        7G  |   text text text text    |
            |   text text text text
            |   text text text text
```

```
           V   text text text text    |
               text text text text    |  G
               text text text text    |
               last line of a file    V
```

Another way to move to a line is using the "%" command with a count. For example, "50%" moves you halfway through the file, and "90%" goes to near the end.

The previous assumes that you want to move to a line in the file, no matter if it's currently visible or not. What if you want to move to one of the lines you can see? This figure shows the three commands you can use:

```
                       +--------------------------+
               H -->   | text sample text         |
                       | sample text              |
                       | text sample text         |
                       | sample text              |
               M -->   | text sample text         |
                       | sample text              |
                       | text sample text         |
                       | sample text              |
               L -->   | text sample text         |
                       +--------------------------+
```

Hints: "H" stands for Home, "M" for Middle and "L" for Last. Alternatively, "H" for High, "M" for Middle and "L" for Low.

03.6 Telling where you are

To see where you are in a file, there are three ways:

1. Use the CTRL-G command. You get a message like this (assuming the 'ruler' option is off):

"usr_03.txt" line 233 of 650 --35%-- col 45-52

   This shows the name of the file you are editing, the line number where the cursor is, the total number of lines, the percentage of the way through the file and the column of the cursor.
   Sometimes you will see a split column number. For example, "col 2-9". This indicates that the cursor is positioned on the second character, but because character one is a tab, occupying eight spaces worth of columns, the screen column is 9.

1. Set the 'number' option. This will display a line number in front of every line:

```
:set number
```

   To switch this off again:

```
:set nonumber
```

   Since 'number' is a boolean option, prepending "no" to its name has the effect of switching it off. A boolean option has only these two values, it is either on or off.
   Vim has many options. Besides the boolean ones there are options with a numerical value and string options. You will see examples of this where they are used.

2. Set the 'ruler' option. This will display the cursor position in the lower right corner of the Vim window:

```
:set ruler
```

Using the 'ruler' option has the advantage that it doesn't take much room, thus there is more space for your text.

03.7 Scrolling around

The CTRL-U command scrolls down half a screen of text. Think of looking through a viewing window at the text and moving this window up by half the height of the window. Thus the window moves up over the text, which is backward in the file. Don't worry if you have a little trouble remembering which end is up. Most users have the same problem.
   The CTRL-D command moves the viewing window down half a screen in the file, thus scrolls the text up half a screen.

```
                                      +----------------+
                                      | some text      |
                                      | some text      |
                                      | some text      |
       +---------------+              | some text      |
       | some text     |  CTRL-U  --> |                |
       |               |              | 123456         |
```

```
        | 123456        |              +----------------+
        | 7890          |
        |               |              +----------------+
        | example       |  CTRL-D -->  | 7890           |
        +---------------+              |                |
                                       | example        |
                                       | example        |
                                       | example        |
                                       | example        |
                                       +----------------+
```

To scroll one line at a time use CTRL-E (scroll up) and CTRL-Y (scroll down). Think of CTRL-E to give you one line Extra. (If you use MS-Windows compatible key mappings CTRL-Y will redo a change instead of scroll.)

To scroll forward by a whole screen (except for two lines) use CTRL-F. To scroll backwards, use CTRL-B. These should be easy to remember: F for Forwards and B for Backwards.

A common issue is that after moving down many lines with "j" your cursor is at the bottom of the screen. You would like to see the context of the line with the cursor. That's done with the "zz" command.

```
        +------------------+             +------------------+
        | earlier text     |             | earlier text     |
        | earlier text     |             | earlier text     |
        | earlier text     |             | earlier text     |
        | earlier text     |   zz  -->   | line with cursor |
        | earlier text     |             | later text       |
        | earlier text     |             | later text       |
        | line with cursor |             | later text       |
        +------------------+             +------------------+
```

The "zt" command puts the cursor line at the top, "zb" at the bottom. There are a few more scrolling commands, see Q_sc. To always keep a few lines of context around the cursor, use the 'scrolloff' option.

03.8 Simple searches

To search for a string, use the "/string" command. To find the word include, for example, use the command:

```
/include
```

You will notice that when you type the "/" the cursor jumps to the last line of the Vim window, like with colon commands. That is where you type the word. You can press the backspace key (backarrow or <BS>) to make corrections. Use the <Left> and <Right> cursor keys when necessary.
Pressing <Enter> executes the command.

> **Note:**
> The characters .*[]^%/\?~$ have special meanings. If you want to use them in a search you must put a \ in front of them. See below.

To find the next occurrence of the same string use the "n" command. Use this to find the first #include after the cursor:

```
/#include
```

And then type "n" several times. You will move to each #include in the text. You can also use a count if you know which match you want. Thus "3n" finds the third match. You can also use a count with "/": "4/the" goes to the fourth match of "the".

The "?" command works like "/" but searches backwards:

```
?word
```

The "N" command repeats the last search the opposite direction. Thus using "N" after a "/" command searches backwards, using "N" after "?" searches forwards.

IGNORING CASE

Normally you have to type exactly what you want to find. If you don't care about upper or lowercase in a word, set the 'ignorecase' option:

```
:set ignorecase
```

If you now search for "word", it will also match "Word" and "WORD". To match case again:

```
:set noignorecase
```

HISTORY

Suppose you do three searches:

```
/one
/two
/three
```

Now let's start searching by typing a simple "/" without pressing <Enter>. If
you press <Up> (the cursor key), Vim puts "/three" on the command line.
Pressing <Enter> at this point searches for three. If you do not press
<Enter>, but press <Up> instead, Vim changes the prompt to "/two". Another
press of <Up> moves you to "/one".
   You can also use the <Down> cursor key to move through the history of
search commands in the other direction.

If you know what a previously used pattern starts with, and you want to use it
again, type that character before pressing <Up>. With the previous example,
you can type "/o<Up>" and Vim will put "/one" on the command line.

The commands starting with ":" also have a history. That allows you to recall
a previous command and execute it again. These two histories are separate.

SEARCHING FOR A WORD IN THE TEXT

Suppose you see the word "TheLongFunctionName" in the text and you want to
find the next occurrence of it. You could type "/TheLongFunctionName", but
that's a lot of typing. And when you make a mistake Vim won't find it.
   There is an easier way: Position the cursor on the word and use the "*"
command. Vim will grab the word under the cursor and use it as the search
string.
   The "#" command does the same in the other direction. You can prepend a
count: "3*" searches for the third occurrence of the word under the cursor.

SEARCHING FOR WHOLE WORDS

If you type "/the" it will also match "there". To only find words that end
in "the" use:

```
/the\>
```

The "\>" item is a special marker that only matches at the end of a word.
Similarly "\<" only matches at the beginning of a word. Thus to search for
the word "the" only:

```
/\<the\>
```

This does not match "there" or "soothe". Notice that the "*" and "#" commands
use these start-of-word and end-of-word markers to only find whole words (you
can use "g*" and "g#" to match partial words).

HIGHLIGHTING MATCHES

While editing a program you see a variable called "nr". You want to check
where it's used. You could move the cursor to "nr" and use the "*" command
and press "n" to go along all the matches.

Vim will highlight all matches. That is a very good way to see where the
variable is used, without the need to type commands.
   To switch this off:

```
:set nohlsearch
```

Then you need to switch it on again if you want to use it for the next search
command:

```
:set hlsearch
```

If you only want to remove the highlighting, use this command:

```
:nohlsearch
```

This doesn't reset the option. Instead, it disables the highlighting. As
soon as you execute a search command, the highlighting will be used again.
Also for the "n" and "N" commands.

TUNING SEARCHES

There are a few options that change how searching works. These are the
essential ones:

```
:set nowrapscan
```

This stops the search at the end of the file. Or, when you are searching
backwards, it stops the search at the start of the file. The 'wrapscan'
option is on by default, thus searching wraps around the end of the file.

```
:set noincsearch
```

This disables the display of the matches while you are still typing your search.

INTERMEZZO

If you like one of the options mentioned before, and set it each time you use Vim, you can put the command in your Vim startup file. Edit the file, for example with:

```
:edit ~/.config/nvim/init.vim
```

Then add a line with the command to set the option, just like you typed it in Vim. Example:

```
Go:set hlsearch<Esc>
```

"G" moves to the end of the file. "o" starts a new line, where you type the ":set" command. You end insert mode with <Esc>. Then write and close the file:

```
ZZ
```

If you now start Vim again, the 'hlsearch' option will already be set.

03.9 Simple search patterns

The Vim editor uses regular expressions to specify what to search for. Regular expressions are an extremely powerful and compact way to specify a search pattern. Unfortunately, this power comes at a price, because regular expressions are a bit tricky to specify.
   In this section we mention only a few essential ones. More about search patterns and commands can be found in chapter 27 usr_27.txt. You can find the full explanation here: pattern.

BEGINNING AND END OF A LINE

The ^ character matches the beginning of a line. On an English-US keyboard you find it above the 6. The pattern "include" matches the word include anywhere on the line. But the pattern "^include" matches the word include only if it is at the beginning of a line.
   The $ character matches the end of a line. Therefore, "was$" matches the word was only if it is at the end of a line.

Let's mark the places where "/the" matches in this example line with "x"s:

```
the solder holding one of the chips melted and the
        xxx                     xxx                 xxx
```

Using "/the$" we find this match:

```
the solder holding one of the chips melted and the
                                                    xxx
```

And with "/^the" we find this one:

```
the solder holding one of the chips melted and the
        xxx
```

You can try searching with "/^the$"; it will only match a single line consisting entirely of "the". White space does matter here, thus if a line contains a space after the word, like "the ", the pattern will not match.

MATCHING ANY SINGLE CHARACTER

The . (dot) character matches any existing character. For example, the pattern "c.m" matches a string whose first character is a c, whose second character is anything, and whose third character is m. Example:

```
We use a computer that became the cummin winter.
                 xxx             xxx      xxx
```

MATCHING SPECIAL CHARACTERS

If you really want to match a dot, you must avoid its special meaning by putting a backslash before it.
   If you search for "ter.", you will find these matches:

```
We use a computer that became the cummin winter.
                     xxxx                          xxxx
```

Searching for "ter\." only finds the second match.

03.10 Using marks

When you make a jump to a position with the "G" command, Vim remembers the position from before this jump. This position is called a mark. To go back where you came from, use this command:

```
``
```

This ` is a backtick or open single-quote character.

If you use the same command a second time you will jump back again. That's because the "``" command is a jump itself, and the position from before this jump is remembered.

Generally, every time you do a command that can move the cursor further than within the same line, this is called a jump. This includes the search commands "/" and "n" (it doesn't matter how far away the match is). But not the character searches with "fx" and "tx" or the word movements "w" and "e".
Also, "j" and "k" are not considered to be a jump, even when you use a count to make them move the cursor quite a long way away.

The "``" command jumps back and forth, between two points. The CTRL-O command jumps to older positions (Hint: O for older). CTRL-I then jumps back to newer positions (Hint: for many common keyboard layouts, I is just next to O). Consider this sequence of commands:

```
33G
/^The
CTRL-O
```

You first jump to line 33, then search for a line that starts with "The". Then with CTRL-O you jump back to line 33. Another CTRL-O takes you back to where you started. If you now use CTRL-I you jump to line 33 again. And to the match for "The" with another CTRL-I.

```
             |  example text   ^             |
        33G  |  example text   |  CTRL-O     | CTRL-I
             |  example text   |             |
             V  line 33 text   ^             V
             |  example text   |             |
       /^The |  example text   |  CTRL-O     | CTRL-I
             V  There you are  |             V
                example text
```

Note:
CTRL-I is the same as <Tab>.

The ":jumps" command gives a list of positions you jumped to. The entry which you used last is marked with a ">".

NAMED MARKS **bookmark**

Vim enables you to place your own marks in the text. The command "ma" marks the place under the cursor as mark a. You can place 26 marks (a through z) in your text. You can't see them, it's just a position that Vim remembers.
To go to a mark, use the command `{mark}, where {mark} is the mark letter. Thus to move to the a mark:

```
`a
```

The command "'mark" (single quotation mark, or apostrophe) moves you to the beginning of the line containing the mark. This differs from the "`mark" command, which also moves you to the marked column.

The marks can be very useful when working on two related parts in a file. Suppose you have some text near the start of the file you need to look at, while working on some text near the end of the file.
Move to the text at the start and place the s (start) mark there:

```
ms
```

Then move to the text you want to work on and put the e (end) mark there:

```
me
```

Now you can move around, and when you want to look at the start of the file, you use this to jump there:

```
's
```

Then you can use '' to jump back to where you were, or 'e to jump to the text you were working on at the end.
There is nothing special about using s for start and e for end, they are just easy to remember.

You can use this command to get a list of marks:

```
:marks
```

You will notice a few special marks. These include:

```
'       The cursor position before doing a jump
"       The cursor position when last editing the file
[       Start of the last change
]       End of the last change
```

Next chapter: usr_04.txt Making small changes

Copyright: see manual-copyright

Making small changes

This chapter shows you several ways of making corrections and moving text around. It teaches you the three basic ways to change text: operator-motion, Visual mode and text objects.

04.1 Operators and motions
04.2 Changing text
04.3 Repeating a change
04.4 Visual mode
04.5 Moving text
04.6 Copying text
04.7 Using the clipboard
04.8 Text objects
04.9 Replace mode
04.10 Conclusion

Next chapter: usr_05.txt Set your settings
Previous chapter: usr_03.txt Moving around
Table of contents: usr_toc.txt

Operators and motions

In chapter 2 you learned the "x" command to delete a single character. And using a count: "4x" deletes four characters.

The "dw" command deletes a word. You may recognize the "w" command as the move word command. In fact, the "d" command may be followed by any motion command, and it deletes from the current location to the place where the cursor winds up.

The "4w" command, for example, moves the cursor over four words. The "d4w" command deletes four words.

To err is human. To really foul up you need a computer.
```
                        ------------------>
                                d4w
```
To err is human. you need a computer.

Vim only deletes up to the position where the motion takes the cursor. That's because Vim knows that you probably don't want to delete the first character of a word. If you use the "e" command to move to the end of a word, Vim guesses that you do want to include that last character:

To err is human. you need a computer.
```
                        -------->
                           d2e
```
To err is human. a computer.

Whether the character under the cursor is included depends on the command you used to move to that character. The reference manual calls this "exclusive" when the character isn't included and "inclusive" when it is.

The "$" command moves to the end of a line. The "d$" command deletes from the cursor to the end of the line. This is an inclusive motion, thus the last character of the line is included in the delete operation:

To err is human. a computer.
```
                       ------------>
                            d$
```
To err is human

There is a pattern here: operator-motion. You first type an operator command. For example, "d" is the delete operator. Then you type a motion command like "4l" or "w". This way you can operate on any text you can move over.

04.2 Changing text

Another operator is "c", change. It acts just like the "d" operator, except it leaves you in Insert mode. For example, "cw" changes a word. Or more specifically, it deletes a word and then puts you in Insert mode.

To err is human
```
           ------->
             c2wbe<Esc>
```
To be human

This "c2wbe<Esc>" contains these bits:

```
        c       the change operator
        2w      move two words (they are deleted and Insert mode started)
        be      insert this text
        <Esc>   back to Normal mode
```

You will have noticed something strange: The space before "human" isn't deleted. There is a saying that for every problem there is an answer that is simple, clear, and wrong. That is the case with the example used here for the "cw" command. The c operator works just like the d operator, with one exception: "cw". It actually works like "ce", change to end of word. Thus the space after the word isn't included. This is an exception that dates back to the old Vi. Since many people are used to it now, the inconsistency has remained in Vim.

MORE CHANGES

Like "dd" deletes a whole line, "cc" changes a whole line. It keeps the existing indent (leading white space) though.

Just like "d$" deletes until the end of the line, "c$" changes until the end of the line. It's like doing "d$" to delete the text and then "a" to start Insert mode and append new text.

SHORTCUTS

Some operator-motion commands are used so often that they have been given a single-letter command:

```
        x  stands for  dl  (delete character under the cursor)
        X  stands for  dh  (delete character left of the cursor)
        D  stands for  d$  (delete to end of the line)
        C  stands for  c$  (change to end of the line)
        s  stands for  cl  (change one character)
        S  stands for  cc  (change a whole line)
```

WHERE TO PUT THE COUNT

The commands "3dw" and "d3w" delete three words. If you want to get really picky about things, the first command, "3dw", deletes one word three times; the command "d3w" deletes three words once. This is a difference without a distinction. You can actually put in two counts, however. For example, "3d2w" deletes two words, repeated three times, for a total of six words.

REPLACING WITH ONE CHARACTER

The "r" command is not an operator. It waits for you to type a character, and will replace the character under the cursor with it. You could do the same with "cl" or with the "s" command, but with "r" you don't have to press <Esc> to get back out of insert mode.

there is somerhing grong here
```
        rT          rt    rw
```

There is something wrong here

Using a count with "r" causes that many characters to be replaced with the same character. Example:

There is something wrong here
```
                          5rx
```

There is something xxxxx here

To replace a character with a line break use "r<Enter>". This deletes one character and inserts a line break. Using a count here only applies to the number of characters deleted: "4r<Enter>" replaces four characters with one line break.

04.3 Repeating a change

The "." command is one of the simplest yet powerful commands in Vim. It repeats the last change. For instance, suppose you are editing an HTML file and want to delete all the <B> tags. You position the cursor on the first < and delete the <B> with the command "df>". You then go to the < of the next

/B> and delete it using the "." command. The "." command executes the last change command (in this case, "df>"). To delete another tag, position the cursor on the < and use the "." command.

To <B>generate</B> a table of <B>contents
```
        f<   find first <     --->
        df>  delete to >         -->
        f<   find next <           --------->
        .    repeat df>                    --->
        f<   find next <                      ------------->
        .    repeat df>                                  -->
```

The "." command works for all changes you make, except for "u" (undo), CTRL-R (redo) and commands that start with a colon (:).

Another example: You want to change the word "four" to "five". It appears several times in your text. You can do this quickly with this sequence of commands:

```
        /four<Enter>    find the first string "four"
        cwfive<Esc>     change the word to "five"
        n               find the next "four"
        .               repeat the change to "five"
        n               find the next "four"
        .               repeat the change
                        etc.
```

04.4 Visual mode

To delete simple items the operator-motion changes work quite well. But often it's not so easy to decide which command will move over the text you want to change. Then you can use Visual mode.

You start Visual mode by pressing "v". You move the cursor over the text you want to work on. While you do this, the text is highlighted. Finally type the operator command.
   For example, to delete from the middle of a word to the middle of another:

This is an examination sample of visual mode

```
                            ---------->
                              velllld
```

This is an example of visual mode

When doing this you don't really have to count how many times you have to press "l" to end up in the right position. You can immediately see what text will be deleted when you press "d".

If at any time you decide you don't want to do anything with the highlighted text, just press <Esc> and Visual mode will stop without doing anything.

SELECTING LINES

If you want to work on whole lines, use "V" to start Visual mode. You will see right away that the whole line is highlighted, without moving around. When you move left or right nothing changes. When you move up or down the selection is extended whole lines at a time.
   For example, select three lines with "Vjj":

```
                        +------------------------+
                        | text more text         |
                     >> | more text more text    | |
       selected lines >> | text text text         | | Vjj
                     >> | text more              | V
                        | more text more         |
                        +------------------------+
```

SELECTING BLOCKS

If you want to work on a rectangular block of characters, use CTRL-V to start Visual mode. This is very useful when working on tables.

```
               name            Q1      Q2      Q3
               pierre          123     455     234
               john            0       90      39
               steve           392     63      334
```

To delete the middle "Q2" column, move the cursor to the "Q" of "Q2". Press CTRL-V to start blockwise Visual mode. Now move the cursor three lines down with "3j" and to the next word with "w". You can see the first character of the last column is included. To exclude it, use "h". Now press "d" and the middle column is gone.

GOING TO THE OTHER SIDE

If you have selected some text in Visual mode, and discover that you need to change the other end of the selection, use the "o" command (Hint: o for other end). The cursor will go to the other end, and you can move the cursor to change where the selection starts. Pressing "o" again brings you back to the other end.

When using blockwise selection, you have four corners. "o" only takes you to one of the other corners, diagonally. Use "O" to move to the other corner in the same line.

Note that "o" and "O" in Visual mode work very differently from Normal mode, where they open a new line below or above the cursor.

04.5 Moving text

When you delete something with "d", "x", or another command, the text is saved. You can paste it back by using the "p" command. (The Vim name for this is put).
   Take a look at how this works. First you will delete an entire line, by putting the cursor on the line you want to delete and typing "dd". Now you move the cursor to where you want to put the line and use the "p" (put) command. The line is inserted on the line below the cursor.

```
        a line          a line        a line
        line 2    dd    line 3    p   line 3
        line 3                        line 2
```

Because you deleted an entire line, the "p" command placed the text line below the cursor. If you delete part of a line (a word, for instance), the "p" command puts it just after the cursor.

Some more boring try text to out commands.

```
                          ---->
                           dw
```

Some more boring text to out commands.

```
                          ------->
                             welp
```

Some more boring text to try out commands.

MORE ON PUTTING

The "P" command puts text like "p", but before the cursor. When you deleted a whole line with "dd", "P" will put it back above the cursor. When you deleted a word with "dw", "P" will put it back just before the cursor.

You can repeat putting as many times as you like. The same text will be used.

You can use a count with "p" and "P". The text will be repeated as many times as specified with the count. Thus "dd" and then "3p" puts three copies of the same deleted line.

SWAPPING TWO CHARACTERS

Frequently when you are typing, your fingers get ahead of your brain (or the other way around?). The result is a typo such as "teh" for "the". Vim makes it easy to correct such problems. Just put the cursor on the e of "teh" and execute the command "xp". This works as follows: "x" deletes the character e and places it in a register. "p" puts the text after the cursor, which is after the h.

```
teh     th     the
         x       p
```

04.6 Copying text

To copy text from one place to another, you could delete it, use "u" to undo the deletion and then "p" to put it somewhere else. There is an easier way: yanking. The "y" operator copies text into a register. Then a "p" command can be used to put it.
Yanking is just a Vim name for copying. The "c" letter was already used for the change operator, and "y" was still available. Calling this operator "yank" made it easier to remember to use the "y" key.

Since "y" is an operator, you use "yw" to yank a word. A count is possible as usual. To yank two words use "y2w". Example:

```
let sqr = LongVariable *
                  ------------->
                        y2w
let sqr = LongVariable *
                                 p
let sqr = LongVariable * LongVariable
```

Notice that "yw" includes the white space after a word. If you don't want this, use "ye".

The "yy" command yanks a whole line, just like "dd" deletes a whole line.

```
        a text line   yy        a text line           a text line
        line 2                  line 2          p     line 2
        last line               last line             a text line
                                                      last line
```

"Y" was originally equivalent to "yank the entire line", as opposed to "D" which is "delete to end of the line". "Y" has thus been remapped to mean "yank to end of the line" to make it consistent with the behavior of "D". Mappings will be covered in later chapters.

04.7 Using the clipboard

If you are using the GUI version of Vim (gvim), you can find the "Copy" item in the "Edit" menu. First select some text with Visual mode, then use the Edit/Copy menu item. The selected text is now copied to the clipboard. You can paste the text in other programs. In Vim itself too.

If you have copied text to the clipboard in another application, you can paste it in Vim with the Edit/Paste menu item. This works in Normal mode and Insert mode. In Visual mode the selected text is replaced with the pasted text.

The "Cut" menu item deletes the text before it's put on the clipboard. The "Copy", "Cut" and "Paste" items are also available in the popup menu (only when there is a popup menu, of course). If your Vim has a toolbar, you can also find these items there.

If you are not using the GUI, or if you don't like using a menu, you have to use another way. You use the normal "y" (yank) and "p" (put) commands, but prepend "* (double-quote star) before it. To copy a line to the clipboard:

```
"*yy
```

To put text from the clipboard back into the text:

```
"*p
```

This only works on versions of Vim that include clipboard support. More about the clipboard can be found in section 09.3 and here: clipboard.

04.8 Text objects

If the cursor is in the middle of a word and you want to delete that word, you need to move back to its start before you can do "dw". There is a simpler way to do this: "daw".

```
this is some example text.
                      daw
this is some text.
```

The "d" of "daw" is the delete operator. "aw" is a text object. Hint: "aw" stands for "A Word". Thus "daw" is "Delete A Word". To be precise, the white space after the word is also deleted (or the white space before the word if at the end of the line).

Using text objects is the third way to make changes in Vim. We already had operator-motion and Visual mode. Now we add operator-text object.

It is very similar to operator-motion, but instead of operating on the text between the cursor position before and after a movement command, the text object is used as a whole. It doesn't matter where in the object the cursor was.

To change a whole sentence use "cis". Take this text:

Hello there. This
is an example. Just
some text.

Move to the start of the second line, on "is an". Now use "cis":

Hello there. Just
some text.

The cursor is in between the blanks in the first line. Now you type the new sentence "Another line.":

Hello there. Another line. Just
some text.

"cis" consists of the "c" (change) operator and the "is" text object. This stands for "Inner Sentence". There is also the "as" ("A Sentence") object. The difference is that "as" includes the white space after the sentence and "is" doesn't. If you would delete a sentence, you want to delete the white space at the same time, thus use "das". If you want to type new text the white space can remain, thus you use "cis".

You can also use text objects in Visual mode. It will include the text object in the Visual selection. Visual mode continues, thus you can do this several times. For example, start Visual mode with "v" and select a sentence with "as". Now you can repeat "as" to include more sentences. Finally you use an operator to do something with the selected sentences.

You can find a long list of text objects here: text-objects.

04.9 Replace mode

The "R" command causes Vim to enter replace mode. In this mode, each character you type replaces the one under the cursor. This continues until you type <Esc>.
In this example you start Replace mode on the first "t" of "text":

This is text.
Rinteresting.<Esc>

This is interesting.

You may have noticed that this command replaced 5 characters in the line with twelve others. The "R" command automatically extends the line if it runs out of characters to replace. It will not continue on the next line.

You can switch between Insert mode and Replace mode with the <Insert> key.

When you use <BS> (backspace) to make a correction, you will notice that the old text is put back. Thus it works like an undo command for the previously typed character.

04.10 Conclusion

The operators, movement commands and text objects give you the possibility to make lots of combinations. Now that you know how they work, you can use N operators with M movement commands to make N * M commands!

You can find a list of operators here: operator.

For example, there are many other ways to delete pieces of text. Here are a few common ones:

```
x       delete character under the cursor (short for "dl")
X       delete character before the cursor (short for "dh")
D       delete from cursor to end of line (short for "d$")
dw      delete from cursor to next start of word
db      delete from cursor to previous start of word
diw     delete word under the cursor (excluding white space)
daw     delete word under the cursor (including white space)
dG      delete until the end of the file
dgg     delete until the start of the file
```

If you use "c" instead of "d" they become change commands. And with "y" you yank the text. And so forth.

There are a few common commands to make changes that didn't fit somewhere else:

~ Change case of the character under the cursor, and move the cursor to the next character. This is not an operator (unless 'tildeop' is set), thus you can't use it with a motion command. It does work in Visual mode, where it changes case for all the selected text.

I Start Insert mode after moving the cursor to the first non-blank in the line.

A Start Insert mode after moving the cursor to the end of the line.

Next chapter: usr_05.txt Set your settings

Copyright: see manual-copyright

VIM USER MANUAL by Bram Moolenaar

Set your settings

Vim can be tuned to work like you want it to. This chapter shows you how to make Vim start with options set to different values. Add plugins to extend Vim's capabilities. Or define your own macros.

05.1 The vimrc file
05.2 Example vimrc contents
05.3 Simple mappings
05.4 Adding a package
05.5 Adding a plugin
05.6 Adding a help file
05.7 The option window
05.8 Often used options

Next chapter: usr_06.txt Using syntax highlighting
Previous chapter: usr_04.txt Making small changes
Table of contents: usr_toc.txt

The vimrc file vimrc-intro

You probably got tired of typing commands that you use very often. To start Vim with all your favorite option settings and mappings, you write them in what is called the init.vim file. Vim executes the commands in this file when it starts up.

If you already have a init.vim file (e.g., when your sysadmin has one setup for you), you can edit it this way:

```
:edit $MYVIMRC
```

If you don't have a vimrc file yet, see init.vim to find out where you can create a vimrc file.

This file is always used and is recommended:

~/.config/nvim/init.vim (Unix and OSX)
~/AppData/Local/nvim/init.vim (Windows)

The vimrc file can contain all the commands that you type after a colon. The simplest ones are for setting options. For example, if you want Vim to always start with the 'ignorecase' option on, add this line your vimrc file:

```
set ignorecase
```

For this new line to take effect you need to exit Vim and start it again. Later you will learn how to do this without exiting Vim.

This chapter only explains the most basic items. For more information on how to write a Vim script file: usr_41.txt.

05.2 Example vimrc contents vimrc_example.vim

In the first chapter was explained how to create a vimrc file.

```
:exe 'edit' stdpath('config').'/init.vim'
```

In this section we will explain the various commands that can be specified in this file. This will give you hints about how to set up your own preferences. Not everything will be explained though. Use the ":help" command to find out more.

```
set backup
```

This tells Vim to keep a backup copy of a file when overwriting it. The backup file will have the same name as the original file with "~" added. See 07.4

```
set history=50
```

Keep 50 commands and 50 search patterns in the history. Use another number if you want to remember fewer or more lines.

```
map Q gq
```

This defines a key mapping. More about that in the next section. This defines the "Q" command to do formatting with the "gq" operator. Otherwise the "Q" command adds a multicursor.

```
vnoremap _g y:exe "grep /" .. escape(@", '\\/') .. "/ *.c *.h"<CR>
```

This mapping yanks the visually selected text and searches for it in C files. This is a complicated mapping. You can see that mappings can be used to do quite complicated things. Still, it is just a sequence of commands that are executed like you typed them.

```
filetype plugin indent on
```

This switches on three very clever mechanisms:
 1. Filetype detection.

     Whenever you start editing a file, Vim will try to figure out what kind of
     file this is. When you edit "main.c", Vim will see the ".c" extension and
     recognize this as a "c" filetype. When you edit a file that starts with
     "#!/bin/sh", Vim will recognize it as a "sh" filetype.
     The filetype detection is used for syntax highlighting and the other two
     items below.
     See filetypes.
 2. Using filetype plugin files

     Many different filetypes are edited with different options. For example,
     when you edit a "c" file, it's very useful to set the 'cindent' option to
     automatically indent the lines. These commonly useful option settings are
     included with Vim in filetype plugins. You can also add your own, see
     write-filetype-plugin.
 3. Using indent files

     When editing programs, the indent of a line can often be computed
     automatically. Vim comes with these indent rules for a number of
     filetypes. See :filetype-indent-on and 'indentexpr'.

restore-cursor last-position-jump

```
augroup RestoreCursor
  autocmd!
  autocmd BufReadPre * autocmd FileType <buffer> ++once
    \ let s:line = line("'\"")
    \ | if s:line >= 1 && s:line <= line("$") && &filetype !~# 'commit'
    \      && index(['xxd', 'gitrebase'], &filetype) == -1
    \      && !&diff
    \ |   execute "normal! g`\""
    \ | endif
augroup END
```

Another autocommand. This time it is used after reading any file. The
complicated stuff after it checks if the '" mark is defined, and jumps to it
if so. It doesn't do that when:
- editing a commit or rebase message, which are likely a different one than

  last time,
- using xxd(1) to filter and edit binary files, which transforms input files

  back and forth, causing them to have dual nature, so to speak (see also
  using-xxd) and
- Vim is in diff mode

The backslash at the start of a line is used to continue the command from the
previous line. That avoids a line getting very long. See line-continuation.
This only works in a Vim script file, not when typing commands at the
command line.

```
command DiffOrig vert new | set bt=nofile | r ++edit # | 0d_
          \ | diffthis | wincmd p | diffthis
```

This adds the ":DiffOrig" command. Use this in a modified buffer to see the
differences with the file it was loaded from. See diff and :DiffOrig.

```
set nolangremap
```

Prevent that the langmap option applies to characters that result from a
mapping. If set (default), this may break plugins (but it's backward
compatible). See 'langremap'.

05.3 Simple mappings

A mapping enables you to bind a set of Vim commands to a single key. Suppose,
for example, that you need to surround certain words with curly braces. In
other words, you need to change a word such as "amount" into "{amount}". With
the :map command, you can tell Vim that the F5 key does this job. The command
is as follows:

```
:map <F5> i{<Esc>ea}<Esc>
```

Note:
When entering this command, you must enter <F5> by typing four characters. Similarly, <Esc> is not entered by pressing the <Esc> key, but by typing five characters. Watch out for this difference when reading the manual!

Let's break this down:

<F5>	The F5 function key. This is the trigger key that causes the command to be executed as the key is pressed.
i{<Esc>	Insert the { character. The <Esc> key ends Insert mode.
e	Move to the end of the word.
a}<Esc>	Append the } to the word.

After you execute the ":map" command, all you have to do to put {} around a word is to put the cursor on the first character and press F5.

In this example, the trigger is a single key; it can be any string. But when you use an existing Vim command, that command will no longer be available. You better avoid that.

One key that can be used with mappings is the backslash. Since you probably want to define more than one mapping, add another character. You could map "\p" to add parentheses around a word, and "\c" to add curly braces, for example:

```
:map \p i(<Esc>ea)<Esc>
:map \c i{<Esc>ea}<Esc>
```

You need to type the \ and the p quickly after another, so that Vim knows they belong together.

The ":map" command (with no arguments) lists your current mappings. At least the ones for Normal mode. More about mappings in section 40.1.

05.4 Adding a package add-package

You may use :packadd to enable packages on demand. This is useful for plugins you want to enable only sometimes. To enable example_package, use the following command:

```
packadd example_package
```

That's all! Now you can find help about this plugin:

```
:help example_package
```

This works, because when :packadd loaded the plugin it also added the package directory in 'runtimepath', so that the help file can be found.

A package is a set of files that you can add to Vim. There are two kinds of packages: optional and automatically loaded on startup.

You can find packages on the Internet in various places. It usually comes as an archive or as a repository. For an archive you can follow these steps:

1. create the package directory:

   ```
   mkdir -p ~/.local/share/nvim/site/pack/fancy
   ```

   "fancy" can be any name of your liking. Use one that describes the package.

2. unpack the archive in that directory. This assumes the top

   directory in the archive is "start":

   ```
   cd ~/.local/share/nvim/site/pack/fancy
   unzip /tmp/fancy.zip
   ```

   If the archive layout is different make sure that you end up with a path like this:

   ```
   ~/.local/share/nvim/site/pack/fancy/start/fancytext/plugin/fancy.vim
   ```

   Here "fancytext" is the name of the package, it can be anything else.

Adding nohlsearch package nohlsearch-install package-nohlsearch

Load the plugin with this command:

```
packadd nohlsearch
```

Automatically execute :nohlsearch after 'updatetime' or getting into Insert mode.
Thus assuming default updatetime, hlsearch would be suspended/turned off after 4 seconds of idle time.

To disable the effect of the plugin after it has been loaded:

```
au! nohlsearch
```

More information about packages can be found here: packages.

05.5 Adding a plugin

add-plugin plugin

Vim's functionality can be extended by adding plugins. A plugin is nothing more than a Vim script file that is loaded automatically when Vim starts. You can add a plugin very easily by dropping it in your plugin directory.

There are two types of plugins:

```
    global plugin: Used for all kinds of files
  filetype plugin: Only used for a specific type of file
```

The global plugins will be discussed first, then the filetype ones add-filetype-plugin.

GLOBAL PLUGINS

When you start Vim, it will automatically load a number of global plugins. You don't have to do anything for this. They add functionality that most people will want to use, but which was implemented as a Vim script instead of being compiled into Vim. You can find them listed in the help index standard-plugin-list. Also see load-plugins.

add-global-plugin

You can add a global plugin to add functionality that will always be present when you use Vim. There are only two steps for adding a global plugin:

1. Get a copy of the plugin.

2. Drop it in the right directory.

GETTING A GLOBAL PLUGIN

Where can you find plugins?

- Some are always loaded, you can see them in the directory

  $VIMRUNTIME/plugin.
- Some come with Vim. You can find them in the directory $VIMRUNTIME/scripts

  and its sub-directories and under $VIM/vimfiles/pack/dist/opt/.
- Download from the net. There is a large collection on https://www.vim.org.

- They are sometimes posted in a Vim maillist.

- You could write one yourself, see write-plugin.

USING A GLOBAL PLUGIN

First read the text in the plugin itself to check for any special conditions. Then copy the file to your plugin directory:

system	plugin directory
Unix	~/.local/share/nvim/site/plugin

Example for Unix (assuming you didn't have a plugin directory yet):

```
mkdir -p ~/.local/share/nvim/site/plugin
cp /tmp/yourplugin.vim ~/.local/share/nvim/site/plugin
```

That's all! Now you can use the commands defined in this plugin.

Instead of putting plugins directly into the plugin/ directory, you may better organize them by putting them into subdirectories under plugin/. As an example, consider using "~/.local/share/nvim/site/plugin/perl/*.vim" for all your Perl plugins.

FILETYPE PLUGINS

add-filetype-plugin ftplugins

The Vim distribution comes with a set of plugins for different filetypes that you can start using with this command:

```
:filetype plugin on
```

That's all! See vimrc-filetype.

If you are missing a plugin for a filetype you are using, or you found a better one, you can add it. There are two steps for adding a filetype plugin:

1. Get a copy of the plugin.

2. Drop it in the right directory.

GETTING A FILETYPE PLUGIN

You can find them in the same places as the global plugins. Watch out if the type of file is mentioned, then you know if the plugin is a global or a filetype one. The scripts in $VIMRUNTIME/scripts are global ones, the filetype plugins are in $VIMRUNTIME/ftplugin.

USING A FILETYPE PLUGIN ftplugin-name

You can add a filetype plugin by dropping it in the right directory. The name of this directory is in the same directory mentioned above for global plugins, but the last part is "ftplugin". Suppose you have found a plugin for the "stuff" filetype, and you are on Unix. Then you can move this file to the ftplugin directory:

```
mkdir -p ~/.local/share/nvim/site/ftplugin
mv thefile ~/.local/share/nvim/site/ftplugin/stuff.vim
```

If that file already exists you already have a plugin for "stuff". You might want to check if the existing plugin doesn't conflict with the one you are adding. If it's OK, you can give the new one another name:

```
mv thefile ~/.local/share/nvim/site/ftplugin/stuff_too.vim
```

The underscore is used to separate the name of the filetype from the rest, which can be anything. If you use "otherstuff.vim" it wouldn't work, it would be loaded for the "otherstuff" filetype.

The generic names for the filetype plugins are:

```
ftplugin/<filetype>.vim
ftplugin/<filetype>_<name>.vim
ftplugin/<filetype>/<name>.vim
```

Here "<name>" can be any name that you prefer.
Examples for the "stuff" filetype on Unix:

```
~/.local/share/nvim/site/ftplugin/stuff.vim
~/.local/share/nvim/site/ftplugin/stuff_def.vim
~/.local/share/nvim/site/ftplugin/stuff/header.vim
```

The <filetype> part is the name of the filetype the plugin is to be used for. Only files of this filetype will use the settings from the plugin. The <name> part of the plugin file doesn't matter, you can use it to have several plugins for the same filetype. Note that it must end in ".vim" or ".lua".

Further reading:

filetype-plugins	Documentation for the filetype plugins and information about how to avoid that mappings cause problems.
load-plugins	When the global plugins are loaded during startup.
ftplugin-overrule	Overruling the settings from a global plugin.
write-plugin	How to write a plugin script.
plugin-details	For more information about using plugins or when your plugin doesn't work.
new-filetype	How to detect a new file type.

05.6 Adding a help file add-local-help

If you are lucky, the plugin you installed also comes with a help file. We will explain how to install the help file, so that you can easily find help for your new plugin.

Let us suppose a plugin ("my-plugin"), which comes with a help file in a non-standard place (it usually resides in a sub-folder called doc/).

First, create a "doc" directory in one of the directories in 'runtimepath':

```
:!mkdir -p ~/.local/share/nvim/site/doc
```

Now, copy the help file to the "doc" directory:

```
:!cp my-plugin/my-plugin-doc.txt ~/.local/share/nvim/site/doc
```

Here comes the trick, which allows you to jump to the subjects in the new help file. Generate the local tags file with the :helptags command:

```
:helptags ~/.local/share/nvim/site/doc
```

You can see an entry for the local help file when you do:

```
:help local-additions
```

The title lines from the local help files are automagically added to this section. There you can see which local help files have been added and jump to them through the tag.

For writing a local help file, see write-local-help.

05.7 The option window

If you are looking for an option that does what you want, you can search in the help files here: options. Another way is by using this command:

```
:options
```

This opens a new window, with a list of options with a one-line explanation. The options are grouped by subject. Move the cursor to a subject and press <Enter> to jump there. Press <Enter> again to jump back. Or use CTRL-O.

You can change the value of an option. For example, move to the "displaying text" subject. Then move the cursor down to this line:

set wrap   nowrap

When you hit <Enter>, the line will change to:

set nowrap   wrap

The option has now been switched off.

Just above this line is a short description of the 'wrap' option. Move the cursor one line up to place it in this line. Now hit <Enter> and you jump to the full help on the 'wrap' option.

For options that take a number or string argument you can edit the value. Then press <Enter> to apply the new value. For example, move the cursor a few lines up to this line:

set so=0

Position the cursor on the zero with "$". Change it into a five with "r5". Then press <Enter> to apply the new value. When you now move the cursor around you will notice that the text starts scrolling before you reach the border. This is what the 'scrolloff' option does, it specifies an offset from the window border where scrolling starts.

05.8 Often used options

There are an awful lot of options. Most of them you will hardly ever use. Some of the more useful ones will be mentioned here. Don't forget you can find more help on these options with the ":help" command, with single quotes before and after the option name. For example:

```
:help 'wrap'
```

In case you have messed up an option value, you can set it back to the default by putting an ampersand (&) after the option name. Example:

```
:set iskeyword&
```

NOT WRAPPING LINES

Vim normally wraps long lines, so that you can see all of the text. Sometimes it's better to let the text continue right of the window. Then you need to scroll the text left-right to see all of a long line. Switch wrapping off with this command:

```
:set nowrap
```

Vim will automatically scroll the text when you move to text that is not displayed. To see a context of ten characters, do this:

```
:set sidescroll=10
```

This doesn't change the text in the file, only the way it is displayed.

WRAPPING MOVEMENT COMMANDS

Most commands for moving around will stop moving at the start and end of a line. You can change that with the 'whichwrap' option. This sets it to the default value:

```
:set whichwrap=b,s
```

This allows the <BS> key, when used in the first position of a line, to move the cursor to the end of the previous line. And the <Space> key moves from the end of a line to the start of the next one.

To allow the cursor keys <Left> and <Right> to also wrap, use this command:

```
:set whichwrap=b,s,<,>
```

This is still only for Normal mode. To let <Left> and <Right> do this in
Insert mode as well:

```
:set whichwrap=b,s,<,>,[,]
```

There are a few other flags that can be added, see 'whichwrap'.

VIEWING TABS

When there are tabs in a file, you cannot see where they are. To make them
visible:

```
:set list
```

Now every tab is displayed as ^I. And a $ is displayed at the end of each
line, so that you can spot trailing spaces that would otherwise go unnoticed.
   A disadvantage is that this looks ugly when there are many Tabs in a file.
If you have a color terminal, or are using the GUI, Vim can show the spaces
and tabs as highlighted characters. Use the 'listchars' option:

```
:set listchars=tab:>-,trail:-
```

Now every tab will be displayed as ">---" (with more or less "-") and trailing
white space as "-". Looks a lot better, doesn't it?

KEYWORDS

The 'iskeyword' option specifies which characters can appear in a word:

```
:set iskeyword
```

iskeyword=@,48-57,_,192-255

The "@" stands for all alphabetic letters. "48-57" stands for ASCII
characters 48 to 57, which are the numbers 0 to 9. "192-255" are the
printable latin characters.
   Sometimes you will want to include a dash in keywords, so that commands
like "w" consider "upper-case" to be one word. You can do it like this:

```
:set iskeyword+=-
:set iskeyword
```

iskeyword=@,48-57,_,192-255,-

If you look at the new value, you will see that Vim has added a comma for you.
   To remove a character use "-=". For example, to remove the underscore:

```
:set iskeyword-=_
:set iskeyword
```

iskeyword=@,48-57,192-255,-

This time a comma is automatically deleted.

ROOM FOR MESSAGES

When Vim starts there is one line at the bottom that is used for messages.
When a message is long, it is either truncated, thus you can only see part of
it, or the text scrolls and you have to press <Enter> to continue.
   You can set the 'cmdheight' option to the number of lines used for
messages. Example:

```
:set cmdheight=3
```

This does mean there is less room to edit text, thus it's a compromise.

Next chapter: usr_06.txt Using syntax highlighting

Copyright: see manual-copyright

Using syntax highlighting

Black and white text is boring. With colors your file comes to life. This not only looks nice, it also speeds up your work. Change the colors used for the different sorts of text. Print your text, with the colors you see on the screen.

06.1 Switching it on
06.2 No or wrong colors?
06.3 Different colors
06.4 With colors or without colors
06.5 Further reading

Next chapter: usr_07.txt Editing more than one file
Previous chapter: usr_05.txt Set your settings
Table of contents: usr_toc.txt

Switching it on

Syntax highlighting is enabled by default. Nvim will automagically detect the type of file and load the right syntax highlighting.

06.2 No or wrong colors?

There can be a number of reasons why you don't see colors:

- Your terminal does not support colors.

  Vim will use bold, italic and underlined text, but this doesn't look very nice. You probably will want to try to get a terminal with colors.
- Your terminal does support colors, but Vim doesn't know this.

  Make sure your $TERM setting is correct. For example, when using an xterm that supports colors:

```
setenv TERM xterm-color
```

  or (depending on your shell):

```
TERM=xterm-color; export TERM
```

  The terminal name must match the terminal you are using.
- The file type is not recognized.

  Vim doesn't know all file types, and sometimes it's near to impossible to tell what language a file uses. Try this command:

```
:set filetype
```

  If the result is "filetype=" then the problem is indeed that Vim doesn't know what type of file this is. You can set the type manually:

```
:set filetype=fortran
```

  To see which types are available, look in the directory $VIMRUNTIME/syntax. For the GUI you can use the Syntax menu.
  Setting the filetype can also be done with a modeline, so that the file will be highlighted each time you edit it. For example, this line can be used in a Makefile (put it near the start or end of the file):

```
# vim: syntax=make
```

  You might know how to detect the file type yourself. Often the file name extension (after the dot) can be used.
  See new-filetype for how to tell Vim to detect that file type.
- There is no highlighting for your file type.

  You could try using a similar file type by manually setting it as mentioned above. If that isn't good enough, you can write your own syntax file, see mysyntaxfile.

Or the colors could be wrong:

- The colored text is very hard to read.

  Vim guesses the background color that you are using. If it is black (or another dark color) it will use light colors for text. If it is white (or another light color) it will use dark colors for text. If

Vim guessed wrong the text will be hard to read. To solve this, set
the 'background' option. For a dark background:

```
:set background=dark
```

And for a light background:

```
:set background=light
```

Make sure you put this _before_ the ":syntax enable" command,
otherwise the colors will already have been set. You could do
":syntax reset" after setting 'background' to make Vim set the default
colors again.

- The colors are wrong when scrolling bottom to top.

Vim doesn't read the whole file to parse the text. It starts parsing
wherever you are viewing the file. That saves a lot of time, but
sometimes the colors are wrong. A simple fix is hitting CTRL-L. Or
scroll back a bit and then forward again.
For a real fix, see :syn-sync. Some syntax files have a way to make
it look further back, see the help for the specific syntax file. For
example, ft-tex-syntax for the TeX syntax.

06.3 Different colors :syn-default-override

If you don't like the default colors, you can select another color scheme. In
the GUI use the Edit/Color Scheme menu. You can also type the command:

```
:colorscheme evening
```

"evening" is the name of the color scheme. There are several others you might
want to try out. Look in the directory $VIMRUNTIME/colors.

When you found the color scheme that you like, add the ":colorscheme" command
to your init.vim file.

You could also write your own color scheme. This is how you do it:

1. Select a color scheme that comes close. Copy this file to your own Vim

   directory. For Unix, this should work:

   ```
   !mkdir -p ~/.config/nvim/colors
   !cp $VIMRUNTIME/colors/morning.vim ~/.config/nvim/colors/mine.vim
   ```

   This is done from Vim, because it knows the value of $VIMRUNTIME.
2. Edit the color scheme file. These entries are useful:

   ```
   cterm           attributes in a color terminal
   ctermfg         foreground color in a color terminal
   ctermbg         background color in a color terminal
   gui             attributes in the GUI
   guifg           foreground color in the GUI
   guibg           background color in the GUI
   ```

   For example, to make comments green:

   ```
   :highlight Comment ctermfg=green guifg=green
   ```

   Attributes you can use for "cterm" and "gui" are "bold" and "underline".
   If you want both, use "bold,underline". For details see the :highlight
   command.
1. Tell Vim to always use your color scheme. Put this line in your vimrc:

   ```
   colorscheme mine
   ```

   If you want to see what the most often used color combinations look like, use
   this command:

   ```
   :runtime syntax/colortest.vim
   ```

   You will see text in various color combinations. You can check which ones are
   readable and look nice.

06.4 With colors or without colors

Displaying text in color takes a lot of effort. If you find the displaying
too slow, you might want to disable syntax highlighting for a moment:

```
:syntax clear
```

When editing another file (or the same one) the colors will come back.

If you want to stop highlighting completely use:

```
:syntax off
```

This will completely disable syntax highlighting and remove it immediately for
all buffers. See :syntax-off for more details.

:syn-manual

If you want syntax highlighting only for specific files, use this:

```
:syntax manual
```

This will enable the syntax highlighting, but not switch it on automatically
when starting to edit a buffer. To switch highlighting on for the current
buffer, set the 'syntax' option:

```
:set syntax=ON
```

06.5 Further reading

usr_44.txt Your own syntax highlighted.
syntax   All the details.

Next chapter: usr_07.txt Editing more than one file

Copyright: see manual-copyright

Editing more than one file

No matter how many files you have, you can edit them without leaving Vim.
Define a list of files to work on and jump from one to the other. Copy text
from one file and put it in another one.

07.1 Edit another file
07.2 A list of files
07.3 Jumping from file to file
07.4 Backup files
07.5 Copy text between files
07.6 Viewing a file
07.7 Changing the file name

Next chapter: usr_08.txt Splitting windows
Previous chapter: usr_06.txt Using syntax highlighting
Table of contents: usr_toc.txt

Edit another file

So far you had to start Vim for every file you wanted to edit. There is a
simpler way. To start editing another file, use this command:

```
:edit foo.txt
```

You can use any file name instead of "foo.txt". Vim will close the current
file and open the new one. If the current file has unsaved changes, however,
Vim displays an error message and does not open the new file:

E37: No write since last change (use ! to override)

> **Note:**
> Vim puts an error ID at the start of each error message. If you do
> not understand the message or what caused it, look in the help system
> for this ID. In this case:

```
:help E37
```

At this point, you have a number of alternatives. You can write the file
using this command:

```
:write
```

Or you can force Vim to discard your changes and edit the new file, using the
force (!) character:

```
:edit! foo.txt
```

If you want to edit another file, but not write the changes in the current
file yet, you can make it hidden:

```
:hide edit foo.txt
```

The text with changes is still there, but you can't see it. This is further
explained in section 22.4: The buffer list.

07.2 A list of files

You can start Vim to edit a sequence of files. For example:

```
vim one.c two.c three.c
```

This command starts Vim and tells it that you will be editing three files.
Vim displays just the first file. After you have done your thing in this
file, to edit the next file you use this command:

```
:next
```

If you have unsaved changes in the current file, you will get an error
message and the ":next" will not work. This is the same problem as with
":edit" mentioned in the previous section. To abandon the changes:

```
:next!
```

But mostly you want to save the changes and move on to the next file. There
is a special command for this:

```
:wnext
```

This does the same as using two separate commands:

```
:write
:next
```

WHERE AM I?

To see which file in the argument list you are editing, look in the window
title. It should show something like "(2 of 3)". This means you are editing
the second file out of three files.
   If you want to see the list of files, use this command:

```
:args
```

This is short for "arguments". The output might look like this:

one.c [two.c] three.c

These are the files you started Vim with. The one you are currently editing,
"two.c", is in square brackets.

MOVING TO OTHER ARGUMENTS

To go back one file:

```
:previous
```

This is just like the ":next" command, except that it moves in the other
direction. Again, there is a shortcut command for when you want to write the
file first:

```
:wprevious
```

To move to the very last file in the list:

```
:last
```

And to move back to the first one again:

```
:first
```

There is no ":wlast" or ":wfirst" command though!

You can use a count for ":next" and ":previous". To skip two files forward:

```
:2next
```

AUTOMATIC WRITING

When moving around the files and making changes, you have to remember to use
":write". Otherwise you will get an error message. If you are sure you
always want to write modified files, you can tell Vim to automatically write
them:

```
:set autowrite
```

When you are editing a file which you may not want to write, switch it off
again:

```
:set noautowrite
```

EDITING ANOTHER LIST OF FILES

You can redefine the list of files without the need to exit Vim and start it
again. Use this command to edit three other files:

```
:args five.c six.c seven.h
```

Or use a wildcard, like it's used in the shell:

```
:args *.txt
```

Vim will take you to the first file in the list. Again, if the current file
has changes, you can either write the file first, or use ":args!" (with !
added) to abandon the changes.

DID YOU EDIT THE LAST FILE?

arglist-quit

When you use a list of files, Vim assumes you want to edit them all. To
protect you from exiting too early, you will get this error when you didn't
edit the last file in the list yet:

E173: 46 more files to edit

If you really want to exit, just do it again. Then it will work (but not when you did other commands in between).

07.3 Jumping from file to file

To quickly jump between two files, press CTRL-^ (on English-US keyboards the ^ is above the 6 key). Example:

```
:args one.c two.c three.c
```

You are now in one.c.

```
:next
```

Now you are in two.c. Now use CTRL-^ to go back to one.c. Another CTRL-^ and you are back in two.c. Another CTRL-^ and you are in one.c again. If you now do:

```
:next
```

You are in three.c. Notice that the CTRL-^ command does not change the idea of where you are in the list of files. Only commands like ":next" and ":previous" do that.

The file you were previously editing is called the "alternate" file. When you just started Vim CTRL-^ will not work, since there isn't a previous file.

PREDEFINED MARKS

After jumping to another file, you can use two predefined marks which are very useful:

```
`"
```

This takes you to the position where the cursor was when you left the file. Another mark that is remembered is the position where you made the last change:

```
`.
```

Suppose you are editing the file "one.txt". Somewhere halfway through the file you use "x" to delete a character. Then you go to the last line with "G" and write the file with ":w". You edit several other files, and then use ":edit one.txt" to come back to "one.txt". If you now use `" Vim jumps to the last line of the file.  Using `. takes you to the position where you deleted the character. Even when you move around in the file " and. will take you to the remembered position. At least until you make another change or leave the file.

FILE MARKS

In section 03.10 was explained how you can place a mark in a file with "mx" and jump to that position with "`x". That works within one file. If you edit another file and place marks there, these are specific for that file. Thus each file has its own set of marks, they are local to the file.
   So far we were using marks with a lowercase letter. There are also marks with an uppercase letter. These are global, they can be used from any file. For example suppose that we are editing the file "foo.txt". Go to halfway down the file ("50%") and place the F mark there (F for foo):

```
50%mF
```

Now edit the file "bar.txt" and place the B mark (B for bar) at its last line:

```
GmB
```

Now you can use the "'F" command to jump back to halfway of foo.txt. Or edit yet another file, type "'B" and you jump to the end of bar.txt.

The file marks are remembered until they are placed somewhere else. Thus you can place the mark, do hours of editing and still be able to jump back to that mark.
   It's often useful to think of a simple connection between the mark letter and where it is placed. For example, use the H mark in a header file, M in a Makefile and C in a C code file.

To see where a specific mark is, give an argument to the ":marks" command:

```
:marks M
```

You can also give several arguments:

```
:marks MCP
```

Don't forget that you can use CTRL-O and CTRL-I to jump to older and newer
positions without placing marks there.

07.4 Backup files

Usually Vim does not produce a backup file. If you want to have one, all you
need to do is execute the following command:

```
:set backup
```

The name of the backup file is the original file with a ~ added to the end.
If your file is named data.txt, for example, the backup file name is
data.txt~.
   If you do not like the fact that the backup files end with ~, you can
change the extension:

```
:set backupext=.bak
```

This will use data.txt.bak instead of data.txt~.
   Another option that matters here is 'backupdir'. It specifies where the
backup file is written. The default, to write the backup in the same
directory as the original file, will mostly be the right thing.

> **Note:**
> When the 'backup' option isn't set but the 'writebackup' is, Vim will
> still create a backup file. However, it is deleted as soon as writing
> the file was completed successfully. This functions as a safety
> against losing your original file when writing fails in some way (disk
> full is the most common cause; being hit by lightning might be
> another, although less common).

KEEPING THE ORIGINAL FILE

If you are editing source files, you might want to keep the file before you
make any changes. But the backup file will be overwritten each time you write
the file. Thus it only contains the previous version, not the first one.
   To make Vim keep the original file, set the 'patchmode' option. This
specifies the extension used for the first backup of a changed file. Usually
you would do this:

```
:set patchmode=.orig
```

When you now edit the file data.txt for the first time, make changes and write
the file, Vim will keep a copy of the unchanged file under the name
"data.txt.orig".
   If you make further changes to the file, Vim will notice that
"data.txt.orig" already exists and leave it alone. Further backup files will
then be called "data.txt~" (or whatever you specified with 'backupext').
   If you leave 'patchmode' empty (that is the default), the original file
will not be kept.

07.5 Copy text between files

This explains how to copy text from one file to another. Let's start with a
simple example. Edit the file that contains the text you want to copy. Move
the cursor to the start of the text and press "v". This starts Visual mode.
Now move the cursor to the end of the text and press "y". This yanks (copies)
the selected text.
   To copy the above paragraph, you would do:

```
:edit thisfile
/This
vjjjj$y
```

Now edit the file you want to put the text in. Move the cursor to the
character where you want the text to appear after. Use "p" to put the text
there.

```
:edit otherfile
/There
p
```

Of course you can use many other commands to yank the text. For example, to
select whole lines start Visual mode with "V". Or use CTRL-V to select a
rectangular block. Or use "yy" to yank a single line, "yaw" to yank-a-word,
etc.
   The "p" command puts the text after the cursor. Use "P" to put the text
before the cursor. Notice that Vim remembers if you yanked a whole line or a
block, and puts it back that way.

USING REGISTERS

When you want to copy several pieces of text from one file to another, having to switch between the files and writing the target file takes a lot of time. To avoid this, copy each piece of text to its own register.
   A register is a place where Vim stores text. Here we will use the registers named a to z (later you will find out there are others). Let's copy a sentence to the f register (f for First):

```
"fyas
```

The "yas" command yanks a sentence like before. It's the "f that tells Vim the text should be placed in the f register. This must come just before the yank command.
   Now yank three whole lines to the l register (l for line):

```
"l3yy
```

The count could be before the "l just as well. To yank a block of text to the b (for block) register:

```
CTRL-Vjjww"by
```

Notice that the register specification "b is just before the "y" command. This is required. If you would have put it before the "w" command, it would not have worked.
   Now you have three pieces of text in the f, l and b registers. Edit another file, move around and place the text where you want it:

```
"fp
```

Again, the register specification "f comes before the "p" command.
   You can put the registers in any order. And the text stays in the register until you yank something else into it. Thus you can put it as many times as you like.

When you delete text, you can also specify a register. Use this to move several pieces of text around. For example, to delete-a-word and write it in the w register:

```
"wdaw
```

Again, the register specification comes before the delete command "d".

APPENDING TO A FILE

When collecting lines of text into one file, you can use this command:

```
:write >> logfile
```

This will write the text of the current file to the end of "logfile". Thus it is appended. This avoids that you have to copy the lines, edit the log file and put them there. Thus you save two steps. But you can only append to the end of a file.
   To append only a few lines, select them in Visual mode before typing ":write". In chapter 10 you will learn other ways to select a range of lines.

07.6 Viewing a file

Sometimes you only want to see what a file contains, without the intention to ever write it back. There is the risk that you type ":w" without thinking and overwrite the original file anyway. To avoid this, edit the file read-only.
   To start Vim in readonly mode, use this command:

```
vim -R file
```

On Unix this command should do the same thing:

```
view file
```

You are now editing "file" in read-only mode. When you try using ":w" you will get an error message and the file won't be written.
   When you try to make a change to the file Vim will give you a warning:

W10: Warning: Changing a readonly file

The change will be done though. This allows for formatting the file, for example, to be able to read it easily.
   If you make changes to a file and forgot that it was read-only, you can still write it. Add the ! to the write command to force writing.

If you really want to forbid making changes in a file, do this:

```
vim -M file
```

Now every attempt to change the text will fail. The help files are like this,
for example. If you try to make a change you get this error message:

E21: Cannot make changes, 'modifiable' is off

You could use the -M argument to setup Vim to work in a viewer mode. This is
only voluntary though, since these commands will remove the protection:

```
:set modifiable
:set write
```

07.7 Changing the file name

A clever way to start editing a new file is by using an existing file that
contains most of what you need. For example, you start writing a new program
to move a file. You know that you already have a program that copies a file,
thus you start with:

```
:edit copy.c
```

You can delete the stuff you don't need. Now you need to save the file under
a new name. The ":saveas" command can be used for this:

```
:saveas move.c
```

Vim will write the file under the given name, and edit that file. Thus the
next time you do ":write", it will write "move.c". "copy.c" remains
unmodified.
   When you want to change the name of the file you are editing, but don't
want to write the file, you can use this command:

```
:file move.c
```

Vim will mark the file as "not edited". This means that Vim knows this is not
the file you started editing. When you try to write the file, you might get
this message:

E13: File exists (use ! to override)

This protects you from accidentally overwriting another file.

Next chapter: usr_08.txt Splitting windows

Copyright: see manual-copyright

VIM USER MANUAL by Bram Moolenaar

Splitting windows

Display two different files above each other. Or view two locations in the file at the same time. See the difference between two files by putting them side by side. All this is possible with split windows.

08.1 Split a window
08.2 Split a window on another file
08.3 Window size
08.4 Vertical splits
08.5 Moving windows
08.6 Commands for all windows
08.7 Viewing differences with diff mode
08.8 Various
08.9 Tabpages

Next chapter: usr_09.txt Using the GUI
Previous chapter: usr_07.txt Editing more than one file
Table of contents: usr_toc.txt

Split a window

The easiest way to open a new window is to use the following command:

```
:split
```

This command splits the screen into two windows and leaves the cursor in the top one:

```
+----------------------------------+
|/* file one.c */                 |
|~                                 |
|~                                 |
|one.c=============================|
|/* file one.c */                 |
|~                                 |
|one.c=============================|
|                                  |
+----------------------------------+
```

What you see here is two windows on the same file. The line with "====" is the status line. It displays information about the window above it. (In practice the status line will be in reverse video.)
   The two windows allow you to view two parts of the same file. For example, you could make the top window show the variable declarations of a program, and the bottom one the code that uses these variables.

The CTRL-W w command can be used to jump between the windows. If you are in the top window, CTRL-W w jumps to the window below it. If you are in the bottom window it will jump to the first window. (CTRL-W CTRL-W does the same thing, in case you let go of the CTRL key a bit later.)

CLOSE THE WINDOW

To close a window, use the command:

```
:close
```

Actually, any command that quits editing a file works, like ":quit" and "ZZ". But ":close" prevents you from accidentally exiting Vim when you close the last window.

CLOSING ALL OTHER WINDOWS

If you have opened a whole bunch of windows, but now want to concentrate on one of them, this command will be useful:

```
:only
```

This closes all windows, except for the current one. If any of the other windows has changes, you will get an error message and that window won't be closed.

08.2 Split a window on another file

The following command opens a second window and starts editing the given file:

```
:split two.c
```

If you were editing one.c, then the result looks like this:

```
+---------------------------------+
|/* file two.c */                 |
|~                                |
|~                                |
|two.c============================|
|/* file one.c */                 |
|~                                |
|one.c============================|
|                                 |
+---------------------------------+
```

To open a window on a new, empty file, use this:

```
:new
```

You can repeat the ":split" and ":new" commands to create as many windows as
you like.

08.3 Window size

The ":split" command can take a number argument. If specified, this will be
the height of the new window. For example, the following opens a new window
three lines high and starts editing the file alpha.c:

```
:3split alpha.c
```

For existing windows you can change the size in several ways. When you have a
working mouse, it is easy: Move the mouse pointer to the status line that
separates two windows, and drag it up or down.

To increase the size of a window:

```
CTRL-W +
```

To decrease it:

```
CTRL-W -
```

Both of these commands take a count and increase or decrease the window size
by that many lines. Thus "4 CTRL-W +" make the window four lines higher.

To set the window height to a specified number of lines:

```
{height}CTRL-W _
```

That's: a number {height}, CTRL-W and then an underscore (the - key with Shift
on English-US keyboards).
   To make a window as high as it can be, use the CTRL-W _ command without a
count.

USING THE MOUSE

In Vim you can do many things very quickly from the keyboard. Unfortunately,
the window resizing commands require quite a bit of typing. In this case,
using the mouse is faster. Position the mouse pointer on a status line. Now
press the left mouse button and drag. The status line will move, thus making
the window on one side higher and the other smaller.

OPTIONS

The 'winheight' option can be set to a minimal desired height of a window and
'winminheight' to a hard minimum height.
   Likewise, there is 'winwidth' for the minimal desired width and
'winminwidth' for the hard minimum width.
   The 'equalalways' option, when set, makes Vim equalize the windows sizes
when a window is closed or opened.

08.4 Vertical splits

The ":split" command creates the new window above the current one. To make
the window appear at the left side, use:

```
:vsplit
```

or:

```
:vsplit two.c
```

The result looks something like this:

```
+-------------------------------------+
|/* file two.c */   |/* file one.c */  |
|~                  |~                 |
|~                  |~                 |
|~                  |~                 |
|two.c===============one.c=============|
|                                     |
+-------------------------------------+
```

Actually, the | lines in the middle will be in reverse video. This is called the vertical separator. It separates the two windows left and right of it.

There is also the ":vnew" command, to open a vertically split window on a new, empty file. Another way to do this:

```
:vertical new
```

The ":vertical" command can be inserted before another command that splits a window. This will cause that command to split the window vertically instead of horizontally. (If the command doesn't split a window, it works unmodified.)

MOVING BETWEEN WINDOWS

Since you can split windows horizontally and vertically as much as you like, you can create almost any layout of windows. Then you can use these commands to move between them:

```
        CTRL-W h        move to the window on the left
        CTRL-W j        move to the window below
        CTRL-W k        move to the window above
        CTRL-W l        move to the window on the right

        CTRL-W t        move to the TOP window
        CTRL-W b        move to the BOTTOM window
```

You will notice the same letters as used for moving the cursor. And the cursor keys can also be used, if you like.
   More commands to move to other windows: Q_wi.

08.5 Moving windows

You have split a few windows, but now they are in the wrong place. Then you need a command to move the window somewhere else. For example, you have three windows like this:

```
+---------------------------------+
|/* file two.c */                 |
|~                                |
|~                                |
|two.c============================|
|/* file three.c */               |
|~                                |
|~                                |
|three.c==========================|
|/* file one.c */                 |
|~                                |
|one.c============================|
|                                 |
+---------------------------------+
```

Clearly the last one should be at the top. Go to that window (using CTRL-W w) and then type this command:

```
CTRL-W K
```

This uses the uppercase letter K. What happens is that the window is moved to the very top. You will notice that K is again used for moving upwards.
   When you have vertical splits, CTRL-W K will move the current window to the top and make it occupy the full width of the Vim window. If this is your layout:

```
+-----------------------------------------+
|/* two.c */  |/* three.c */  |/* one.c */  |
|~            |~              |~           |
|~            |~              |~           |
|~            |~              |~           |
|~            |~              |~           |
|~            |~              |~           |
|two.c=========three.c=========one.c========|
|                                         |
+-----------------------------------------+
```

Then using CTRL-W K in the middle window (three.c) will result in:

```
+-----------------------------------------+
|/* three.c */                            |
|~                                        |
|~                                        |
|three.c==================================|
|/* two.c */           |/* one.c */        |
|~                     |~                 |
|two.c==================one.c=============|
|                                         |
+-----------------------------------------+
```

The other three similar commands (you can probably guess these now):

```
        CTRL-W H        move window to the far left
        CTRL-W J        move window to the bottom
        CTRL-W L        move window to the far right
```

08.6 Commands for all windows

When you have several windows open and you want to quit Vim, you can close each window separately. A quicker way is using this command:

```
:qall
```

This stands for "quit all". If any of the windows contain changes, Vim will not exit. The cursor will automatically be positioned in a window with changes. You can then either use ":write" to save the changes, or ":quit!" to throw them away.

If you know there are windows with changes, and you want to save all these changes, use this command:

```
:wall
```

This stands for "write all". But actually, it only writes files with changes. Vim knows it doesn't make sense to write files that were not changed.
   And then there is the combination of ":qall" and ":wall": the "write and quit all" command:

```
:wqall
```

This writes all modified files and quits Vim.
   Finally, there is a command that quits Vim and throws away all changes:

```
:qall!
```

Be careful, there is no way to undo this command!

OPENING A WINDOW FOR ALL ARGUMENTS

To make Vim open a window for each file, start it with the "-o" argument:

```
vim -o one.txt two.txt three.txt
```

This results in:

```
+-------------------------------+
|file one.txt                   |
|~                              |
|one.txt========================|
|file two.txt                   |
|~                              |
|two.txt========================|
|file three.txt                 |
|~                              |
|three.txt======================|
|                               |
+-------------------------------+
```

The "-O" argument is used to get vertically split windows.
   When Vim is already running, the ":all" command opens a window for each file in the argument list. ":vertical all" does it with vertical splits.

08.7 Viewing differences with diff mode

There is a special way to start Nvim, which shows the differences between two files. Let's take a file "main.c" and insert a few characters in one line. Write this file with the 'backup' option set, so that the backup file "main.c~" will contain the previous version of the file.
Type this command in a shell to start Nvim in diff mode:

```
nvim -d main.c~ main.c
```

Vim will start, with two windows side by side. You will only see the line
in which you added characters, and a few lines above and below it.

```
 VV                   VV
+----------------------------------------+
|+ +--123 lines: /* a|+ +--123 lines: /* a|  <- fold
|  text              |  text              |
|  text              |  text              |
|  text              |  text              |
|  text              |  changed text      |  <- changed line
|  text              |  text              |
|  text              |  ------------------|  <- deleted line
|  text              |  text              |
|  text              |  text              |
|  text              |  text              |
|+ +--432 lines: text|+ +--432 lines: text|  <- fold
|  ~                 |  ~                 |
|  ~                 |  ~                 |
|main.c~==============main.c==============|
|                                        |
+----------------------------------------+
```

(This picture doesn't show the highlighting, use "nvim -d" for that.)

The lines that were not modified have been collapsed into one line. This is
called a closed fold. They are indicated in the picture with "<- fold". Thus
the single fold line at the top stands for 123 text lines. These lines are
equal in both files.
   The line marked with "<- changed line" is highlighted, and the inserted
text is displayed with another color. This clearly shows what the difference
is between the two files.
   The line that was deleted is displayed with "---" in the main.c window.
See the "<- deleted line" marker in the picture. These characters are not
really there. They just fill up main.c, so that it displays the same number
of lines as the other window.

THE FOLD COLUMN

Each window has a column on the left with a slightly different background. In
the picture above these are indicated with "VV". You notice there is a plus
character there, in front of each closed fold. Move the mouse pointer to that
plus and click the left button. The fold will open, and you can see the text
that it contains.
   The fold column contains a minus sign for an open fold. If you click on
this -, the fold will close.
   Obviously, this only works when you have a working mouse. You can also use
"zo" to open a fold and "zc" to close it.

DIFFING IN VIM

Another way to start in diff mode can be done from inside Vim. Edit the
"main.c" file, then make a split and show the differences:

```
:edit main.c
:vertical diffsplit main.c~
```

The ":vertical" command is used to make the window split vertically. If you
omit this, you will get a horizontal split.

If you have a patch or diff file, you can use the third way to start diff
mode. First edit the file to which the patch applies. Then tell Vim the name
of the patch file:

```
:edit main.c
:vertical diffpatch main.c.diff
```

WARNING: The patch file must contain only one patch, for the file you are
editing. Otherwise you will get a lot of error messages, and some files might
be patched unexpectedly.
   The patching will only be done to the copy of the file in Vim. The file on
your harddisk will remain unmodified (until you decide to write the file).

SCROLL BINDING

When the files have more changes, you can scroll in the usual way. Vim will
try to keep both the windows start at the same position, so you can easily see
the differences side by side.
   When you don't want this for a moment, use this command:

```
:set noscrollbind
```

JUMPING TO CHANGES

When you have disabled folding in some way, it may be difficult to find the
changes. Use this command to jump forward to the next change:

```
]c
```

To go the other way use:

```
[c
```

Prepend a count to jump further away. Thus "4]c" jumps to the fourth next
change, and "3[c" jumps to the third previous change.

REMOVING CHANGES

You can move text from one window to the other. This either removes
differences or adds new ones. Vim doesn't keep the highlighting updated in
all situations. To update it use this command:

```
:diffupdate
```

To remove a difference, you can move the text in a highlighted block from one
window to another. Take the "main.c" and "main.c~" example above. Move the
cursor to the left window, on the line that was deleted in the other window.
Now type this command:

```
dp
```

The change will be removed by putting the text of the current window in the
other window. "dp" stands for "diff put".
   You can also do it the other way around. Move the cursor to the right
window, to the line where "changed" was inserted. Now type this command:

```
do
```

The change will now be removed by getting the text from the other window.
Since there are no changes left now, Vim puts all text in a closed fold.
"do" stands for "diff obtain". "dg" would have been better, but that already
has a different meaning ("dgg" deletes from the cursor until the first line).

For details about diff mode, see diff-mode.

08.8 Various

The 'laststatus' option can be used to specify when the last window has a
statusline:

```
        0       never
        1       only when there are split windows (the default)
        2       always
        3       have a global statusline at the bottom instead of one for each
                window
```

Many commands that edit another file have a variant that splits the window.
For Command-line commands this is done by prepending an "s". For example:
":tag" jumps to a tag, ":stag" splits the window and jumps to a
tag.
   For Normal mode commands a CTRL-W is prepended. CTRL-^ jumps to the
alternate file, CTRL-W CTRL-^ splits the window and edits the alternate file.

The 'splitbelow' option can be set to make a new window appear below the
current window. The 'splitright' option can be set to make a vertically split
window appear right of the current window.

When splitting a window you can prepend a modifier command to tell where the
window is to appear:

```
        :leftabove {cmd}        left or above the current window
        :aboveleft {cmd}        idem
        :rightbelow {cmd}       right or below the current window
        :belowright {cmd}       idem
        :topleft {cmd}          at the top or left of the Vim window
        :botright {cmd}         at the bottom or right of the Vim window
```

08.9 Tabpages

You will have noticed that windows never overlap. That means you quickly run
out of screen space. The solution for this is called Tabpages.

Assume you are editing "thisfile". To create a new tabpage use this command:

```
:tabedit thatfile
```

This will edit the file "thatfile" in a window that occupies the whole Vim
window. And you will notice a bar at the top with the two file names:

```
+----------------------------------+
| thisfile | /thatfile/ __________X|    (thatfile is bold)
|/* thatfile */                    |
|that                              |
|that                              |
|~                                 |
|~                                 |
|~                                 |
|                                  |
+----------------------------------+
```

You now have two tabpages. The first one has a window for "thisfile" and the second one a window for "thatfile". It's like two pages that are on top of each other, with a tab sticking out of each page showing the file name.

Now use the mouse to click on "thisfile" in the top line. The result is

```
+----------------------------------+
| /thisfile/ | thatfile __________X|    (thisfile is bold)
|/* thisfile */                    |
|this                              |
|this                              |
|~                                 |
|~                                 |
|~                                 |
|                                  |
+----------------------------------+
```

Thus you can switch between tabpages by clicking on the label in the top line. If you don't have a mouse or don't want to use it, you can use the "gt" command. Mnemonic: Goto Tab.

Now let's create another tabpage with the command:

```
:tab split
```

This makes a new tabpage with one window that is editing the same buffer as the window we were in:

```
+-----------------------------------+
| thisfile | /thisfile/ | thatfile __X|   (thisfile is bold)
|/* thisfile */                     |
|this                               |
|this                               |
|~                                  |
|~                                  |
|~                                  |
|                                   |
+-----------------------------------+
```

You can put ":tab" before any Ex command that opens a window. The window will be opened in a new tabpage. Another example:

```
:tab help gt
```

Will show the help text for "gt" in a new tabpage.

A few more things you can do with tabpages:

- click with the mouse in the space after the last label

  The next tabpage will be selected, like with "gt".
- click with the mouse on the "X" in the top right corner

  The current tabpage will be closed. Unless there are unsaved changes in the current tabpage.
- double click with the mouse in the top line

  A new tabpage will be created.
- the "tabonly" command

  Closes all tabpages except the current one. Unless there are unsaved changes in other tabpages.

For more information about tabpages see tabpage.

Next chapter: usr_09.txt Using the GUI

Copyright: see manual-copyright

Using the GUI

Vim works in an ordinary terminal, while gVim has a Graphical User Interface (GUI). It can do the same things and a few more. The GUI offers menus, a toolbar, scrollbars and other items. This chapter is about these extra things that the GUI offers.

09.1 Parts of the GUI
09.2 Using the mouse
09.3 The clipboard
09.4 Select mode

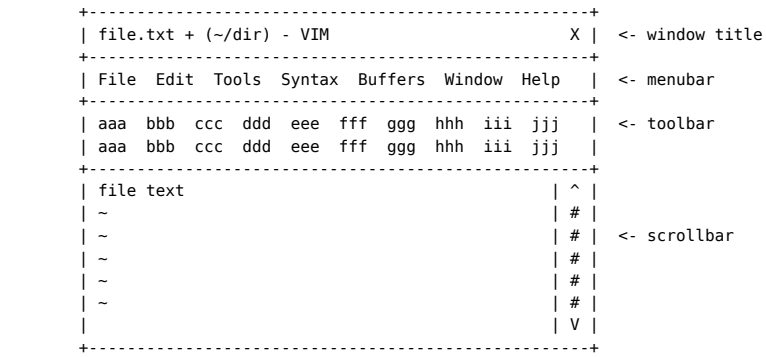
Next chapter: usr_10.txt Making big changes
Previous chapter: usr_08.txt Splitting windows
Table of contents: usr_toc.txt

Parts of the GUI

You might have an icon on your desktop that starts gvim. Otherwise, one of these commands should do it:

```
gvim file.txt
vim -g file.txt
```

If this doesn't work you don't have a version of Vim with GUI support. You will have to install one first.
Vim will open a window and display "file.txt" in it. What the window looks like depends on the version of Vim. It should resemble the following picture (for as far as this can be shown in ASCII!).



The largest space is occupied by the file text. This shows the file in the same way as in a terminal. With some different colors and another font perhaps.

THE WINDOW TITLE

At the very top is the window title. This is drawn by your window system. Vim will set the title to show the name of the current file. First comes the name of the file. Then some special characters and the directory of the file in parens. These special characters can be present:

- - The file cannot be modified (e.g., a help file)
- + The file contains changes
- = The file is read-only
- =+ The file is read-only, contains changes anyway

If nothing is shown you have an ordinary, unchanged file.

THE MENUBAR

You know how menus work, right? Vim has the usual items, plus a few more. Browse them to get an idea of what you can use them for. A relevant submenu is Edit/Global Settings. You will find these entries:

```
Toggle Toolbar          make the toolbar appear/disappear
Toggle Bottom Scrollbar make a scrollbar appear/disappear at the bottom
Toggle Left Scrollbar   make a scrollbar appear/disappear at the left
Toggle Right Scrollbar  make a scrollbar appear/disappear at the right
```

THE TOOLBAR

This contains icons for the most often used actions. Hopefully the icons are self-explanatory. There are tooltips to get an extra hint (move the mouse pointer to the icon without clicking and don't move it for a second).

The "Edit/Global Settings/Toggle Toolbar" menu item can be used to make the toolbar disappear. If you never want a toolbar, use this command in your

vimrc file:

```
:set guioptions-=T
```

This removes the 'T' flag from the 'guioptions' option. Other parts of the GUI can also be enabled or disabled with this option. See the help for it.

THE SCROLLBARS

By default there is one scrollbar on the right. It does the obvious thing. When you split the window, each window will get its own scrollbar.
   You can make a horizontal scrollbar appear with the menu item Edit/Global Settings/Toggle Bottom Scrollbar. This is useful in diff mode, or when the 'wrap' option has been reset (more about that later).

When there are vertically split windows, only the windows on the right side will have a scrollbar. However, when you move the cursor to a window on the left, it will be this one that the scrollbar controls. This takes a bit of time to get used to.
   When you work with vertically split windows, consider adding a scrollbar on the left. This can be done with a menu item, or with the 'guioptions' option:

```
:set guioptions+=l
```

This adds the 'l' flag to 'guioptions'.

09.2 Using the mouse

Standards are wonderful. In Microsoft Windows, you can use the mouse to select text in a standard manner. The X Window system also has a standard system for using the mouse. Unfortunately, these two standards are not the same. Fortunately, you can customize Vim.

The following commands makes the mouse work more like a Microsoft Windows mouse:

```
set selection=exclusive
set selectmode=mouse,key
set keymodel=startsel,stopsel
```

The mouse can be further tuned. Check out these options if you want to change the way how the mouse works:

'mouse'	in which mode the mouse is used by Vim
'mousemodel'	what effect a mouse click has
'mousetime'	time between clicks for a double-click
'mousehide'	hide the mouse while typing
'selectmode'	whether the mouse starts Visual or Select mode

09.3 The clipboard

In section 04.7 the basic use of the clipboard was explained. There is one essential thing to explain about X-windows: There are actually two places to exchange text between programs. MS-Windows doesn't have this.

In X-Windows there is the "current selection". This is the text that is currently highlighted. In Vim this is the Visual area (this assumes you are using the default option settings). You can paste this selection in another application without any further action.
   For example, in this text select a few words with the mouse. Vim will switch to Visual mode and highlight the text. Now start another gvim, without a file name argument, so that it displays an empty window. Click the middle mouse button. The selected text will be inserted.

The "current selection" will only remain valid until some other text is selected. After doing the paste in the other gvim, now select some characters in that window. You will notice that the words that were previously selected in the other gvim window are displayed differently. This means that it no longer is the current selection.

You don't need to select text with the mouse, using the keyboard commands for Visual mode works just as well.

THE REAL CLIPBOARD

Now for the other place with which text can be exchanged. We call this the "real clipboard", to avoid confusion. Often both the "current selection" and the "real clipboard" are called clipboard, you'll have to get used to that.
   To put text on the real clipboard, select a few different words in one of the gvims you have running. Then use the Edit/Copy menu entry. Now the text has been copied to the real clipboard. You can't see this, unless you have some application that shows the clipboard contents (e.g., KDE's Klipper).
   Now select the other gvim, position the cursor somewhere and use the Edit/Paste menu. You will see the text from the real clipboard is inserted.

USING BOTH

This use of both the "current selection" and the "real clipboard" might sound a bit confusing. But it is very useful. Let's show this with an example. Use one gvim with a text file and perform these actions:

- Select two words in Visual mode.
- Use the Edit/Copy menu to get these words onto the clipboard.
- Select one other word in Visual mode.
- Use the Edit/Paste menu item. What will happen is that the single selected word is replaced with the two words from the clipboard.
- Move the mouse pointer somewhere else and click the middle button. You will see that the word you just overwrote with the clipboard is inserted here.

If you use the "current selection" and the "real clipboard" with care, you can do a lot of useful editing with them.

USING THE KEYBOARD

If you don't like using the mouse, you can access the current selection and the real clipboard with two registers. The "* register is for the current selection.
   To make text become the current selection, use Visual mode. For example, to select a whole line just press "V".
   To insert the current selection before the cursor:

```
"*P
```

Notice the uppercase "P". The lowercase "p" puts the text after the cursor.

The "+ register is used for the real clipboard. For example, to copy the text from the cursor position until the end of the line to the clipboard:

```
"+y$
```

Remember, "y" is yank, which is Vim's copy command.
   To insert the contents of the real clipboard before the cursor:

```
"+P
```

It's the same as for the current selection, but uses the plus (+) register instead of the star "*" register.

09.4 Select mode

And now something that is used more often on MS-Windows than on X-Windows. But both can do it. You already know about Visual mode. Select mode is like Visual mode, because it is also used to select text. But there is an obvious difference: When typing text, the selected text is deleted and the typed text replaces it.

To start working with Select mode, you must first enable it (for MS-Windows it is probably already enabled, but you can do this anyway):

```
:set selectmode+=mouse
```

Now use the mouse to select some text. It is highlighted like in Visual mode. Now press a letter. The selected text is deleted, and the single letter replaces it. You are in Insert mode now, thus you can continue typing.

Since typing normal text causes the selected text to be deleted, you can not use the normal movement commands "hjkl", "w", etc. Instead, use the shifted function keys. <S-Left> (shifted cursor left key) moves the cursor left. The selected text is changed like in Visual mode. The other shifted cursor keys do what you expect. <S-End> and <S-Home> also work.

You can tune the way Select mode works with the 'selectmode' option.

Next chapter: usr_10.txt Making big changes

Copyright: see manual-copyright

Making big changes

In chapter 4 several ways to make small changes were explained. This chapter goes into making changes that are repeated or can affect a large amount of text. The Visual mode allows doing various things with blocks of text. Use an external program to do really complicated things.

10.1 Record and playback commands
10.2 Substitution
10.3 Command ranges
10.4 The global command
10.5 Visual block mode
10.6 Reading and writing part of a file
10.7 Formatting text
10.8 Changing case
10.9 Using an external program

Next chapter: usr_11.txt Recovering from a crash
Previous chapter: usr_09.txt Using the GUI
Table of contents: usr_toc.txt

Record and playback commands

The "." command repeats the preceding change. But what if you want to do something more complex than a single change? That's where command recording comes in. There are three steps:

1. The "q{register}" command starts recording keystrokes into the register named {register}. The register name must be between a and z.
2. Type your commands.
3. To finish recording, press q (without any extra character).

You can now execute the macro by typing the command "@{register}".

Take a look at how to use these commands in practice. You have a list of filenames that look like this:

stdio.h
fcntl.h
unistd.h
stdlib.h

And what you want is the following:

#include "stdio.h"
#include "fcntl.h"
#include "unistd.h"
#include "stdlib.h"

You start by moving to the first character of the first line. Next you execute the following commands:

```
qa                      Start recording a macro in register a.
^                       Move to the beginning of the line.
i#include "<Esc>        Insert the string #include " at the beginning
                        of the line.
$                       Move to the end of the line.
a"<Esc>                 Append the character double quotation mark (")
                        to the end of the line.
j                       Go to the next line.
q                       Stop recording the macro.
```

Now that you have done the work once, you can repeat the change by typing the command "@a" three times.

The "@a" command can be preceded by a count, which will cause the macro to be executed that number of times. In this case you would type:

```
3@a
```

MOVE AND EXECUTE

You might have the lines you want to change in various places. Just move the cursor to each location and use the "@a" command. If you have done that once, you can do it again with "@@". That's a bit easier to type. If you now execute register b with "@b", the next "@@" will use register b.

If you compare the playback method with using ".", there are several differences. First of all, "." can only repeat one change. As seen in the example above, "@a" can do several changes, and move around as well. Secondly, "." can only remember the last change. Executing a register allows you to make any changes and then still use "@a" to replay the recorded commands. Finally, you can use 26 different registers. Thus you can remember 26 different command sequences to execute.

USING REGISTERS

The registers used for recording are the same ones you used for yank and delete commands. This allows you to mix recording with other commands to

manipulate the registers.
   Suppose you have recorded a few commands in register n. When you execute this with "@n" you notice you did something wrong. You could try recording again, but perhaps you will make another mistake. Instead, use this trick:

```
G                       Go to the end of the file.
o<Esc>                  Create an empty line.
"np                     Put the text from the n register.  You now see
                        the commands you typed as text in the file.
{edits}                 Change the commands that were wrong.  This is
                        just like editing text.
0                       Go to the start of the line.
"ny$                    Yank the corrected commands into the n
                        register.
dd                      Delete the scratch line.
```

Now you can execute the corrected commands with "@n". (If your recorded commands include line breaks, adjust the last two items in the example to include all the lines.)

APPENDING TO A REGISTER

So far we have used a lowercase letter for the register name. To append to a register, use an uppercase letter.
   Suppose you have recorded a command to change a word to register c. It works properly, but you would like to add a search for the next word to change. This can be done with:

```
qC/word<Enter>q
```

You start with "qC", which records to the c register and appends. Thus writing to an uppercase register name means to append to the register with the same letter, but lowercase.

This works both with recording and with yank and delete commands. For example, you want to collect a sequence of lines into the a register. Yank the first line with:

```
"ayy
```

Now move to the second line, and type:

```
"Ayy
```

Repeat this command for all lines. The a register now contains all those lines, in the order you yanked them.

10.2 Substitution find-replace

The ":substitute" command enables you to perform string replacements on a whole range of lines. The general form of this command is as follows:

```
:[range]substitute/from/to/[flags]
```

This command changes the "from" string to the "to" string in the lines specified with [range]. For example, you can change "Professor" to "Teacher" in all lines with the following command:

```
:%substitute/Professor/Teacher/
```

> **Note:**
> The ":substitute" command is almost never spelled out completely. Most of the time, people use the abbreviated version ":s". From here on the abbreviation will be used.

The "%" before the command specifies the command works on all lines. Without a range, ":s" only works on the current line. More about ranges in the next section 10.3.

By default, the ":substitute" command changes only the first occurrence on each line. For example, the preceding command changes the line:

Professor Smith criticized Professor Johnson today.

to:

Teacher Smith criticized Professor Johnson today.

To change every occurrence on the line, you need to add the g (global) flag. The command:

```
:%s/Professor/Teacher/g
```

results in (starting with the original line):

Teacher Smith criticized Teacher Johnson today.

Other flags include p (print), which causes the ":substitute" command to print out the last line it changes. The c (confirm) flag tells ":substitute" to ask you for confirmation before it performs each substitution. Enter the following:

```
:%s/Professor/Teacher/c
```

Vim finds the first occurrence of "Professor" and displays the text it is about to change. You get the following prompt:

```
replace with Teacher? (y)es/(n)o/(a)ll/(q)uit/(l)ast/scroll up(^E)/down(^Y)
```

At this point, you must enter one of the following answers:

y	Yes; make this change.
n	No; skip this match.
a	All; make this change and all remaining ones without further confirmation.
q	Quit; don't make any more changes.
l	Last; make this change and then quit.
CTRL-E	Scroll the text one line up.
CTRL-Y	Scroll the text one line down.

The "from" part of the substitute command is actually a pattern. The same kind as used for the search command. For example, this command only substitutes "the" when it appears at the start of a line:

```
:s/^the/these/
```

If you are substituting with a "from" or "to" part that includes a slash, you need to put a backslash before it. A simpler way is to use another character instead of the slash. A plus, for example:

```
:s+one/two+one or two+
```

10.3 Command ranges

The ":substitute" command, and many other : commands, can be applied to a selection of lines. This is called a range.
   The simple form of a range is {number},{number}. For example:

```
:1,5s/this/that/g
```

Executes the substitute command on the lines 1 to 5. Line 5 is included. The range is always placed before the command.

A single number can be used to address one specific line:

```
:54s/President/Fool/
```

Some commands work on the whole file when you do not specify a range. To make them work on the current line the "." address is used. The ":write" command works like that. Without a range, it writes the whole file. To make it write only the current line into a file:

```
:.write otherfile
```

The first line always has number one. How about the last line? The "$" character is used for this. For example, to substitute in the lines from the cursor to the end:

```
:.,$s/yes/no/
```

The "%" range that we used before, is actually a short way to say "1,$", from the first to the last line.

USING A PATTERN IN A RANGE

Suppose you are editing a chapter in a book, and want to replace all occurrences of "grey" with "gray". But only in this chapter, not in the next one. You know that only chapter boundaries have the word "Chapter" in the first column. This command will work then:

```
:?^Chapter?,/^Chapter/s=grey=gray=g
```

You can see a search pattern is used twice. The first "?^Chapter?" finds the line above the current position that matches this pattern. Thus the ?pattern? range is used to search backwards. Similarly, "/^Chapter/" is used to search forward for the start of the next chapter.
   To avoid confusion with the slashes, the "=" character was used in the substitute command here. A slash or another character would have worked as well.

ADD AND SUBTRACT

There is a slight error in the above command: If the title of the next chapter had included "grey" it would be replaced as well. Maybe that's what you wanted, but what if you didn't? Then you can specify an offset.
To search for a pattern and then use the line above it:

```
/Chapter/-1
```

You can use any number instead of the 1. To address the second line below the match:

```
/Chapter/+2
```

The offsets can also be used with the other items in a range. Look at this one:

```
:.+3,$-5
```

This specifies the range that starts three lines below the cursor and ends five lines before the last line in the file.

USING MARKS

Instead of figuring out the line numbers of certain positions, remembering them and typing them in a range, you can use marks.
Place the marks as mentioned in chapter 3. For example, use "mt" to mark the top of an area and "mb" to mark the bottom. Then you can use this range to specify the lines between the marks (including the lines with the marks):

```
:'t,'b
```

VISUAL MODE AND RANGES

You can select text with Visual mode. If you then press ":" to start a colon command, you will see this:

```
:'<,'>
```

Now you can type the command and it will be applied to the range of lines that was visually selected.

> **Note:**
> When using Visual mode to select part of a line, or using CTRL-V to select a block of text, the colon commands will still apply to whole lines. This might change in a future version of Vim.

The '< and '> are actually marks, placed at the start and end of the Visual selection. The marks remain at their position until another Visual selection is made. Thus you can use the "'<" command to jump to position where the Visual area started. And you can mix the marks with other items:

```
:'>,$
```

This addresses the lines from the end of the Visual area to the end of the file.

A NUMBER OF LINES

When you know how many lines you want to change, you can type the number and then ":". For example, when you type "5:", you will get:

```
:.,.+4
```

Now you can type the command you want to use. It will use the range "." (current line) until ".+4" (four lines down). Thus it spans five lines.

See also :range, for an overview of all possible ways to specify a range.

10.4 The global command

The ":global" command is one of the more powerful features of Vim. It allows you to find a match for a pattern and execute a command there. The general form is:

```
:[range]global/{pattern}/{command}
```

This is similar to the ":substitute" command. But, instead of replacing the matched text with other text, the command {command} is executed.

> **Note:**
> The command executed for ":global" must be one that starts with a colon. Normal mode commands can not be used directly. The :normal command can do this for you.

Suppose you want to change "foobar" to "barfoo", but only in C++ style comments. These comments start with "//". Use this command:

```
:g+//+s/foobar/barfoo/g
```

This starts with ":g". That is short for ":global", just like ":s" is short for ":substitute". Then the pattern, enclosed in plus characters. Since the pattern we are looking for contains a slash, this uses the plus character to separate the pattern. Next comes the substitute command that changes "foobar" into "barfoo".

The default range for the global command is the whole file. Thus no range was specified in this example. This is different from ":substitute", which works on one line without a range.

The command isn't perfect, since it also matches lines where "//" appears halfway through a line, and the substitution will also take place before the "//".

Just like with ":substitute", any pattern can be used. When you learn more complicated patterns later, you can use them here.

10.5 Visual block mode

With CTRL-V you can start selection of a rectangular area of text. There are a few commands that do something special with the text block.

There is something special about using the "$" command in Visual block mode. When the last motion command used was "$", all lines in the Visual selection will extend until the end of the line, also when the line with the cursor is shorter. This remains effective until you use a motion command that moves the cursor horizontally. Thus using "j" keeps it, "h" stops it.

INSERTING TEXT

The command "I{string}<Esc>" inserts the text {string} in each line, just left of the visual block. You start by pressing CTRL-V to enter visual block mode. Now you move the cursor to define your block. Next you type I to enter Insert mode, followed by the text to insert. As you type, the text appears on the first line only.

After you press <Esc> to end the insert, the text will magically be inserted in the rest of the lines contained in the visual selection. Example:

include one
include two
include three
include four

Move the cursor to the "o" of "one" and press CTRL-V. Move it down with "3j" to "four". You now have a block selection that spans four lines. Now type:

```
Imain.<Esc>
```

The result:

include main.one
include main.two
include main.three
include main.four

If the block spans short lines that do not extend into the block, the text is not inserted in that line. For example, make a Visual block selection that includes the word "long" in the first and last line of this text, and thus has no text selected in the second line:

This is a long line
short
Any other long line
^^^^ selected block

Now use the command "Ivery <Esc>". The result is:

This is a very long line
short
Any other very long line

In the short line no text was inserted.

If the string you insert contains a newline, the "I" acts just like a Normal insert command and affects only the first line of the block.

The "A" command works the same way, except that it appends after the right side of the block. And it does insert text in a short line. Thus you can make a choice whether you do or don't want to append text to a short line.

There is one special case for "A": Select a Visual block and then use "$" to make the block extend to the end of each line. Using "A" now will append the text to the end of each line.

Using the same example from above, and then typing "$A XXX<Esc>, you get this result:

This is a long line XXX
short XXX
Any other long line XXX

This really requires using the "$" command. Vim remembers that it was used. Making the same selection by moving the cursor to the end of the longest line with other movement commands will not have the same result.

CHANGING TEXT

The Visual block "c" command deletes the block and then throws you into Insert mode to enable you to type in a string. The string will be inserted in each line in the block.

Starting with the same selection of the "long" words as above, then typing "c_LONG_<Esc>", you get this:

This is a _LONG_ line
short
Any other _LONG_ line

Just like with "I" the short line is not changed. Also, you can't enter a newline in the new text.

The "C" command deletes text from the left edge of the block to the end of line. It then puts you in Insert mode so that you can type in a string, which is added to the end of each line.

Starting with the same text again, and typing "Cnew text<Esc>" you get:

This is a new text
short
Any other new text

Notice that, even though only the "long" word was selected, the text after it is deleted as well. Thus only the location of the left edge of the visual block really matters.

Again, short lines that do not reach into the block are excluded.

Other commands that change the characters in the block:

```
~       swap case       (a -> A and A -> a)
U       make uppercase  (a -> A and A -> A)
u       make lowercase  (a -> a and A -> a)
```

FILLING WITH A CHARACTER

To fill the whole block with one character, use the "r" command. Again, starting with the same example text from above, and then typing "rx":

This is a xxxx line
short
Any other xxxx line

> **Note:**
> If you want to include characters beyond the end of the line in the block, check out the 'virtualedit' feature in chapter 25.

SHIFTING

The command ">" shifts the selected text to the right one shift amount, inserting whitespace. The starting point for this shift is the left edge of the visual block.

With the same example again, ">" gives this result:

This is a long line
short
Any other long line

The shift amount is specified with the 'shiftwidth' option. To change it to use 4 spaces:

```
:set shiftwidth=4
```

The "<" command removes one shift amount of whitespace at the left edge of the block. This command is limited by the amount of text that is there; so if there is less than a shift amount of whitespace available, it removes what it can.

JOINING LINES

The "J" command joins all selected lines together into one line. Thus it removes the line breaks. Actually, the line break, leading white space and trailing white space is replaced by one space. Two spaces are used after a line ending (that can be changed with the 'joinspaces' option).

Let's use the example that we got so familiar with now. The result of using the "J" command:

This is a long line short Any other long line

The "J" command doesn't require a blockwise selection. It works with "v" and "V" selection in exactly the same way.

If you don't want the white space to be changed, use the "gJ" command.

10.6 Reading and writing part of a file

When you are writing an e-mail message, you may want to include another file. This can be done with the ":read {filename}" command. The text of the file is put below the cursor line.

Starting with this text:

Hi John,
Here is the diff that fixes the bug:
Bye, Pierre.

Move the cursor to the second line and type:

```
:read patch
```

The file named "patch" will be inserted, with this result:

```
Hi John,
Here is the diff that fixes the bug:
2c2
<       for (i = 0; i <= length; ++i)
---
>       for (i = 0; i < length; ++i)
Bye, Pierre.
```

The ":read" command accepts a range. The file will be put below the last line number of this range. Thus ":$r patch" appends the file "patch" at the end of the file.
   What if you want to read the file above the first line? This can be done with the line number zero. This line doesn't really exist, you will get an error message when using it with most commands. But this command is allowed:

```
:0read patch
```

The file "patch" will be put above the first line of the file.

WRITING A RANGE OF LINES

To write a range of lines to a file, the ":write" command can be used. Without a range it writes the whole file. With a range only the specified lines are written:

```
:.,$write tempo
```

This writes the lines from the cursor until the end of the file into the file "tempo". If this file already exists you will get an error message. Vim protects you from accidentally overwriting an existing file. If you know what you are doing and want to overwrite the file, append !:

```
:.,$write! tempo
```

CAREFUL: The ! must follow the ":write" command immediately, without white space. Otherwise it becomes a filter command, which is explained later in this chapter.

APPENDING TO A FILE

In the first section of this chapter was explained how to collect a number of lines into a register. The same can be done to collect lines in a file. Write the first line with this command:

```
:.write collection
```

Now move the cursor to the second line you want to collect, and type this:

```
:.write >>collection
```

The ">>" tells Vim the "collection" file is not to be written as a new file, but the line must be appended at the end.   You can repeat this as many times as you like.

10.7 Formatting text

When you are typing plain text, it's nice if the length of each line is automatically trimmed to fit in the window. To make this happen while inserting text, set the 'textwidth' option:

```
:set textwidth=78
```

You might remember that in the example vimrc file this command was used for every text file. Thus if you are using that vimrc file, you were already using it. To check the current value of 'textwidth':

```
:set textwidth
```

Now lines will be broken to take only up to 78 characters. However, when you insert text halfway through a line, or when you delete a few words, the lines will get too long or too short. Vim doesn't automatically reformat the text. To tell Vim to format the current paragraph:

```
gqap
```

This starts with the "gq" command, which is an operator. Following is "ap", the text object that stands for "a paragraph". A paragraph is separated from the next paragraph by an empty line.

Note:
A blank line, which contains white space, does NOT separate
paragraphs. This is hard to notice!

Instead of "ap" you could use any motion or text object. If your paragraphs
are properly separated, you can use this command to format the whole file:

```
ggggqG
```

"gg" takes you to the first line, "gq" is the format operator and "G" the
motion that jumps to the last line.

In case your paragraphs aren't clearly defined, you can format just the lines
you manually select. Move the cursor to the first line you want to format.
Start with the command "gqj". This formats the current line and the one below
it. If the first line was short, words from the next line will be appended.
If it was too long, words will be moved to the next line. The cursor moves to
the second line. Now you can use "." to repeat the command. Keep doing this
until you are at the end of the text you want to format.

10.8 Changing case

You have text with section headers in lowercase. You want to make the word
"section" all uppercase. Do this with the "gU" operator. Start with the
cursor in the first column:

```
gUw
```

```
        section header      ---->      SECTION header
```

The "gu" operator does exactly the opposite:

```
guw
```

```
        SECTION header      ---->      section header
```

You can also use "g~" to swap case. All these are operators, thus they work
with any motion command, with text objects and in Visual mode.
   To make an operator work on lines you double it. The delete operator is
"d", thus to delete a line you use "dd". Similarly, "gugu" makes a whole line
lowercase. This can be shortened to "guu". "gUgU" is shortened to "gUU" and
"g~g~" to "g~~". Example:

```
g~~
```

Some GIRLS have Fun   ---->   sOME girls HAVE fUN

10.9 Using an external program

Vim has a very powerful set of commands, it can do anything. But there may
still be something that an external command can do better or faster.
   The command "!{motion}{program}" takes a block of text and filters it
through an external program. In other words, it runs the system command
represented by {program}, giving it the block of text represented by {motion}
as input. The output of this command then replaces the selected block.
   Because this summarizes badly if you are unfamiliar with Unix filters, take
a look at an example. The sort command sorts a file. If you execute the
following command, the unsorted file input.txt will be sorted and written to
output.txt. This works on both Unix and Windows.

```
sort <input.txt >output.txt
```

Now do the same thing in Vim. You want to sort lines 1 through 5 of a file.
You start by putting the cursor on line 1. Next you execute the following
command:

```
!5G
```

The "!" tells Vim that you are performing a filter operation. The Vim editor
expects a motion command to follow, indicating which part of the file to
filter. The "5G" command tells Vim to go to line 5, so it now knows that it
is to filter lines 1 (the current line) through 5.
   In anticipation of the filtering, the cursor drops to the bottom of the
screen and a ! prompt displays. You can now type in the name of the filter
program, in this case "sort". Therefore, your full command is as follows:

```
!5Gsort<Enter>
```

The result is that the sort program is run on the first 5 lines. The output
of the program replaces these lines.

```
        line 55                     line 11
        line 33                     line 22
        line 11         -->         line 33
        line 22                     line 44
        line 44                     line 55
        last line                   last line
```

The "!!" command filters the current line through a filter. In Unix the "date" command prints the current time and date. "!!date<Enter>" replaces the current line with the output of "date". This is useful to add a timestamp to a file.

Note: There is a difference between "!cmd" (e.g. using it without any file range) and "{range}!cmd". While the former will simply execute the external command and Vim will show the output, the latter will filter {range}lines through the filter and replace that range by the result of the filter command. See :! and :range! for details.

WHEN IT DOESN'T WORK

Starting a shell, sending it text and capturing the output requires that Vim knows how the shell works exactly. When you have problems with filtering, check the values of these options:

'shell'	specifies the program that Vim uses to execute external programs.
'shellcmdflag'	argument to pass a command to the shell
'shellquote'	quote to be used around the command
'shellxquote'	quote to be used around the command and redirection
'shellslash'	use forward slashes in the command (only for MS-Windows and alikes)
'shellredir'	string used to write the command output into a file

On Unix this is hardly ever a problem, because there are two kinds of shells: "sh" like and "csh" like. Vim checks the 'shell' option and sets related options automatically, depending on whether it sees "csh" somewhere in 'shell'.
   On MS-Windows, however, there are many different shells and you might have to tune the options to make filtering work. Check the help for the options for more information.

READING COMMAND OUTPUT

To read the contents of the current directory into the file, use this:

on Unix:

```
:read !ls
```

on MS-Windows:

```
:read !dir
```

The output of the "ls" or "dir" command is captured and inserted in the text, below the cursor. This is similar to reading a file, except that the "!" is used to tell Vim that a command follows.
   The command may have arguments. And a range can be used to tell where Vim should put the lines:

```
:0read !date -u
```

This inserts the current time and date in UTC format at the top of the file. (Well, if you have a date command that accepts the "-u" argument.) Note the difference with using "!!date": that replaced a line, while ":read !date" will insert a line.

WRITING TEXT TO A COMMAND

The Unix command "wc" counts words. To count the words in the current file:

```
:write !wc
```

This is the same write command as before, but instead of a file name the "!" character is used and the name of an external command. The written text will be passed to the specified command as its standard input. The output could look like this:

4   47   249

The "wc" command isn't verbose. This means you have 4 lines, 47 words and 249 characters.

Watch out for this mistake:

```
:write! wc
```

This will write the file "wc" in the current directory, with force. White space is important here!

REDRAWING THE SCREEN

If the external command produced an error message, the display may have been messed up. Vim is very efficient and only redraws those parts of the screen that it knows need redrawing. But it can't know about what another program has written. To tell Vim to redraw the screen:

```
CTRL-L
```

Next chapter: usr_11.txt Recovering from a crash
Copyright: see manual-copyright

VIM USER MANUAL by Bram Moolenaar

Recovering from a crash

Did your computer crash? And you just spent hours editing? Don't panic! Vim stores enough information to be able to restore most of your work. This chapter shows you how to get your work back and explains how the swap file is used.

11.1 Basic recovery
11.2 Where is the swap file?
11.3 Crashed or not?
11.4 Further reading

Next chapter: usr_12.txt Clever tricks
Previous chapter: usr_10.txt Making big changes
Table of contents: usr_toc.txt

Basic recovery

In most cases recovering a file is quite simple, assuming you know which file you were editing (and the harddisk is still working). Start Vim on the file, with the "-r" argument added:

```
vim -r help.txt
```

Vim will read the swap file (used to store text you were editing) and may read bits and pieces of the original file. If Vim recovered your changes you will see these messages (with different file names, of course):

Using swap file ".help.txt.swp"
Original file "~/vim/runtime/doc/help.txt"
Recovery completed. You should check if everything is OK.
(You might want to write out this file under another name
and run diff with the original file to check for changes)
You may want to delete the .swp file now.

To be on the safe side, write this file under another name:

```
:write help.txt.recovered
```

Compare the file with the original file to check if you ended up with what you expected. Diff mode is very useful for this 08.7. For example:

```
:write help.txt.recovered
:edit #
:diffsp help.txt
```

Watch out for the original file to contain a more recent version (you saved the file just before the computer crashed). And check that no lines are missing (something went wrong that Vim could not recover).
   If Vim produces warning messages when recovering, read them carefully. This is rare though.

If the recovery resulted in text that is exactly the same as the file contents, you will get this message:

Using swap file ".help.txt.swp"
Original file "~/vim/runtime/doc/help.txt"
Recovery completed. Buffer contents equals file contents.
You may want to delete the .swp file now.

This usually happens if you already recovered your changes, or you wrote the file after making changes. It is safe to delete the swap file now.

It is normal that the last few changes can not be recovered. Vim flushes the changes to disk when you don't type for about four seconds, or after typing about two hundred characters. This is set with the 'updatetime' and 'updatecount' options. Thus when Vim didn't get a chance to save itself when the system went down, the changes after the last flush will be lost.

If you were editing without a file name, give an empty string as argument:

```
vim -r ""
```

You must be in the right directory, otherwise Vim can't find the swap file.

11.2 Where is the swap file?

Vim can store the swap file in several places. To find it, change to the directory of the file, and use:

```
vim -r
```

Vim will list the swap files that it can find. It will also look in other directories where the swap file for files in the current directory may be located. It will not find swap files in any other directories though, it doesn't search the directory tree.
   The output could look like this:

Swap files found:

In current directory:
 1.   .main.c.swp ~

                owned by: mool   dated: Tue May 29 21:00:25 2001 ~
               file name: ~mool/vim/vim6/src/main.c ~
                modified: YES ~
               user name: mool   host name: masaka.moolenaar.net ~
              process ID: 12525 ~
          In directory ~/tmp: ~
             -- none -- ~
          In directory /var/tmp: ~
             -- none -- ~
          In directory /tmp: ~
             -- none -- ~

If there are several swap files that look like they may be the one you want to use, a list is given of these swap files and you are requested to enter the number of the one you want to use. Carefully look at the dates to decide which one you want to use.
   In case you don't know which one to use, just try them one by one and check the resulting files if they are what you expected.

USING A SPECIFIC SWAP FILE

If you know which swap file needs to be used, you can recover by giving the swap file name. Vim will then find out the name of the original file from the swap file.

Example:

```
vim -r .help.txt.swo
```

This is also handy when the swap file is in another directory than expected. Vim recognizes files with the pattern "*.s[uvw][a-z]" as swap files.

If this still does not work, see what file names Vim reports and rename the files accordingly. Check the 'directory' option to see where Vim may have put the swap file.

> **Note:**
> Vim tries to find the swap file by searching the directories in the 'dir' option, looking for files that match "filename.sw?". If wildcard expansion doesn't work (e.g., when the 'shell' option is invalid), Vim does a desperate try to find the file "filename.swp". If that fails too, you will have to give the name of the swapfile itself to be able to recover the file.

11.3 Crashed or not? ATTENTION E325

Vim tries to protect you from doing stupid things. Suppose you innocently start editing a file, expecting the contents of the file to show up. Instead, Vim produces a very long message:

E325: ATTENTION
Found a swap file by the name ".main.c.swp"
owned by: mool   dated: Tue May 29 21:09:28 2001
file name: ~mool/vim/vim6/src/main.c
modified: no
user name: mool   host name: masaka.moolenaar.net
process ID: 12559 (still running)
While opening file "main.c"
dated: Tue May 29 19:46:12 2001
 ~
(1) Another program may be editing the same file.
If this is the case, be careful not to end up with two
different instances of the same file when making changes.
Quit, or continue with caution.
 ~
(2) An edit session for this file crashed.
If this is the case, use ":recover" or "vim -r main.c"
to recover the changes (see ":help recovery").
If you did this already, delete the swap file ".main.c.swp"
to avoid this message.

You get this message, because, when starting to edit a file, Vim checks if a swap file already exists for that file. If there is one, there must be something wrong. It may be one of these two situations.

 1. Another edit session is active on this file. Look in the message for the

     line with "process ID". It might look like this:

process ID: 12559 (still running)

   The text "(still running)" indicates that the process editing this file runs on the same computer. When working on a non-Unix system you will not get this extra hint. When editing a file over a network, you may not see the hint, because the process might be running on another computer. In those two cases you must find out what the situation is yourself.
      If there is another Vim editing the same file, continuing to edit will result in two versions of the same file. The one that is written last will

overwrite the other one, resulting in loss of changes. You better quit this Vim.

1. The swap file might be the result from a previous crash of Vim or the computer. Check the dates mentioned in the message. If the date of the swap file is newer than the file you were editing, and this line appears:

modified: YES

Then you very likely have a crashed edit session that is worth recovering. If the date of the file is newer than the date of the swap file, then either it was changed after the crash (perhaps you recovered it earlier, but didn't delete the swap file?), or else the file was saved before the crash but after the last write of the swap file (then you're lucky: you don't even need that old swap file). Vim will warn you for this with this extra line:

NEWER than swap file!

NOTE that in the following situation Vim knows the swap file is not useful and will automatically delete it:

- The file is a valid swap file (Magic number is correct).
- The flag that the file was modified is not set.
- The process is not running.

You can programmatically deal with this situation with the FileChangedShell autocommand event.

UNREADABLE SWAP FILE

Sometimes the line

[cannot be read]

will appear under the name of the swap file. This can be good or bad, depending on circumstances.

It is good if a previous editing session crashed without having made any changes to the file. Then a directory listing of the swap file will show that it has zero bytes. You may delete it and proceed.

It is slightly bad if you don't have read permission for the swap file. You may want to view the file read-only, or quit. On multi-user systems, if you yourself did the last changes under a different login name, a logout followed by a login under that other name might cure the "read error". Or else you might want to find out who last edited (or is editing) the file and have a talk with them.

It is very bad if it means there is a physical read error on the disk containing the swap file. Fortunately, this almost never happens.
You may want to view the file read-only at first (if you can), to see the extent of the changes that were "forgotten". If you are the one in charge of that file, be prepared to redo your last changes.

WHAT TO DO? swap-exists-choices

If dialogs are supported you will be asked to select one of six choices:

Swap file ".main.c.swp" already exists!
[O]pen Read-Only, (E)dit anyway, (R)ecover, (Q)uit, (A)bort, (D)elete it:

O Open the file readonly. Use this when you just want to view the file and don't need to recover it. You might want to use this when you know someone else is editing the file, but you just want to look in it and not make changes.

E Edit the file anyway. Use this with caution! If the file is being edited in another Vim, you might end up with two versions of the file. Vim will try to warn you when this happens, but better be safe than sorry.

R Recover the file from the swap file. Use this if you know that the swap file contains changes that you want to recover.

Q Quit. This avoids starting to edit the file. Use this if there is another Vim editing the same file.
When you just started Vim, this will exit Vim. When starting Vim with files in several windows, Vim quits only if there is a swap file for the first one. When using an edit command, the file will not be loaded and you are taken back to the previously edited file.

A Abort. Like Quit, but also abort further commands. This is useful when loading a script that edits several files, such as a session with multiple windows.

D Delete the swap file. Use this when you are sure you no longer need it. For example, when it doesn't contain changes, or when the file itself is newer than the swap file.
On Unix this choice is only offered when the process that created the swap file does not appear to be running.

If you do not get the dialog (you are running a version of Vim that does not support it), you will have to do it manually. To recover the file, use this command:

```
:recover
```

Vim cannot always detect that a swap file already exists for a file. This is the case when the other edit session puts the swap files in another directory or when the path name for the file is different when editing it on different machines. Therefore, don't rely on Vim always warning you.

If you really don't want to see this message, you can add the 'A' flag to the 'shortmess' option. But it's very unusual that you need this.

For programmatic access to the swap file, see swapinfo().

11.4 Further reading

swap-file	An explanation about where the swap file will be created and what its name is.
:preserve	Manually flushing the swap file to disk.
:swapname	See the name of the swap file for the current file.
'updatecount'	Number of key strokes after which the swap file is flushed to disk.
'updatetime'	Timeout after which the swap file is flushed to disk.
'directory'	List of directory names where to store the swap file.

Next chapter: usr_12.txt Clever tricks

Copyright: see manual-copyright

Clever tricks

By combining several commands you can make Vim do nearly everything. In this chapter a number of useful combinations will be presented. This uses the commands introduced in the previous chapters and a few more.

12.1 Replace a word
12.2 Change "Last, First" to "First Last"
12.3 Sort a list
12.4 Reverse line order
12.5 Count words
12.6 Find a man page
12.7 Trim blanks
12.8 Find where a word is used

Next chapter: usr_20.txt Typing command-line commands quickly
Previous chapter: usr_11.txt Recovering from a crash
Table of contents: usr_toc.txt

Replace a word

The substitute command can be used to replace all occurrences of a word with another word:

```
:%s/four/4/g
```

The "%" range means to replace in all lines. The "g" flag at the end causes all words in a line to be replaced.
This will not do the right thing if your file also contains "thirtyfour". It would be replaced with "thirty4". To avoid this, use the "\<" item to match the start of a word:

```
:%s/\<four/4/g
```

Obviously, this still goes wrong on "fourteen". Use "\>" to match the end of a word:

```
:%s/\<four\>/4/g
```

If you are programming, you might want to replace "four" in comments, but not in the code. Since this is difficult to specify, add the "c" flag to have the substitute command prompt you for each replacement:

```
:%s/\<four\>/4/gc
```

REPLACING IN SEVERAL FILES

Suppose you want to replace a word in more than one file. You could edit each file and type the command manually. It's a lot faster to use record and playback.
Let's assume you have a directory with C++ files, all ending in ".cpp". There is a function called "GetResp" that you want to rename to "GetAnswer".

vim *.cpp	Start Vim, defining the argument list to contain all the C++ files. You are now in the first file.
qq	Start recording into the q register
:%s/\<GetResp\>/GetAnswer/g	Do the replacements in the first file.
:wnext	Write this file and move to the next one.
q	Stop recording.
@q	Execute the q register. This will replay the substitution and ":wnext". You can verify that this doesn't produce an error message.
999@q	Execute the q register on the remaining files.

At the last file you will get an error message, because ":wnext" cannot move to the next file. This stops the execution, and everything is done.

Note:
When playing back a recorded sequence, an error stops the execution. Therefore, make sure you don't get an error message when recording.

There is one catch: If one of the .cpp files does not contain the word "GetResp", you will get an error and replacing will stop. To avoid this, add the "e" flag to the substitute command:

```
:%s/\<GetResp\>/GetAnswer/ge
```

The "e" flag tells ":substitute" that not finding a match is not an error.

12.2 Change "Last, First" to "First Last"

You have a list of names in this form:

Doe, John
Smith, Peter

You want to change that to:

John Doe
Peter Smith

This can be done with just one command:

```
:%s/\([^,]*\), \(.*\)/\2 \1/
```

Let's break this down in parts. Obviously it starts with a substitute command. The "%" is the line range, which stands for the whole file. Thus the substitution is done in every line in the file.
   The arguments for the substitute command are "/from/to/". The slashes separate the "from" pattern and the "to" string. This is what the "from" pattern contains:

```
\([^,]*\), \(.*\)
```

```
       The first part between \( \) matches "Last"     \(     \)
           match anything but a comma                    [^,]
           any number of times                               *
       matches ", " literally                                    ,
       The second part between \( \) matches "First"               \(  \)
           any character                                             .
           any number of times                                        *
```

In the "to" part we have "\2" and "\1". These are called backreferences. They refer to the text matched by the "\(\)" parts in the pattern. "\2" refers to the text matched by the second "\(\)", which is the "First" name. "\1" refers to the first "\(\)", which is the "Last" name.
   You can use up to nine backreferences in the "to" part of a substitute command. "\0" stands for the whole matched pattern. There are a few more special items in a substitute command, see sub-replace-special.

12.3 Sort a list

In a Makefile you often have a list of files. For example:

**OBJS = **
**version.o **
**pch.o **
**getopt.o **
**util.o **
**getopt1.o **
**inp.o **
**patch.o **
backup.o

To sort this list, filter the text through the external sort command:

```
/^OBJS
j
:.,/^$/-1!sort
```

This goes to the first line, where "OBJS" is the first thing in the line. Then it goes one line down and filters the lines until the next empty line. You could also select the lines in Visual mode and then use "!sort". That's easier to type, but more work when there are many lines.
   The result is this:

**OBJS = **
backup.o
**getopt.o **
**getopt1.o **
**inp.o **
**patch.o **
**pch.o **
**util.o **
**version.o **

Notice that a backslash at the end of each line is used to indicate the line continues. After sorting, this is wrong! The "backup.o" line that was at the end didn't have a backslash. Now that it sorts to another place, it must have a backslash.
   The simplest solution is to add the backslash with "A \<Esc>". You can keep the backslash in the last line, if you make sure an empty line comes after it. That way you don't have this problem again.

12.4 Reverse line order

The :global command can be combined with the :move command to move all the lines before the first line, resulting in a reversed file. The command is:

```
:global/^/move 0
```

Abbreviated:

```
:g/^/m 0
```

The "^" regular expression matches the beginning of the line (even if the line is blank). The :move command moves the matching line to after the imaginary zeroth line, so the current matching line becomes the first line of the file. As the :global command is not confused by the changing line numbering, :global proceeds to match all remaining lines of the file and puts each as the first.

This also works on a range of lines. First move to above the first line and mark it with "mt". Then move the cursor to the last line in the range and type:

```
:'t+1,.g/^/m 't
```

12.5 Count words

Sometimes you have to write a text with a maximum number of words. Vim can count the words for you.
When the whole file is what you want to count the words in, use this command:

```
g CTRL-G
```

Do not type a space after the g, this is just used here to make the command easy to read.
The output looks like this:

Col 1 of 0; Line 141 of 157; Word 748 of 774; Byte 4489 of 4976

You can see on which word you are (748), and the total number of words in the file (774).

When the text is only part of a file, you could move to the start of the text, type "g CTRL-G", move to the end of the text, type "g CTRL-G" again, and then use your brain to compute the difference in the word position. That's a good exercise, but there is an easier way. With Visual mode, select the text you want to count words in. Then type g CTRL-G. The result:

Selected 5 of 293 Lines; 70 of 1884 Words; 359 of 10928 Bytes

For other ways to count words, lines and other items, see count-items.

12.6 Find a man page find-manpage

While editing a C program, you are using a function that you want to find the man page for (this is on Unix). Let's first use a simple way: Move the cursor to the word you want to find help on and press

```
K
```

Nvim will run :Man on the word. If the man page is found, it is displayed. You can also use the :Man command to open a window on a man page:

```
:Man csh
```

You can scroll around and the text is highlighted. This allows you to find the help you were looking for. Use CTRL-W w to jump to the window with the text you were working on.
To find a man page in a specific section, put the section number first. For example, to look in section 3 for "echo":

```
:Man 3 echo
```

To jump to another man page, which is in the text with the typical form "word(1)", press CTRL-] on it. Further ":Man" commands will use the same window.

To display a man page for the word under the cursor, use this:

```
K
```

For example, you want to know the return value of "strstr()" while editing this line:

if (strstr (input, "aap") ==)

Move the cursor to somewhere on "strstr" and type "K". A window will open to display the man page for strstr().

12.7 Trim blanks

Some people find spaces and tabs at the end of a line useless, wasteful, and ugly. To remove whitespace at the end of every line, execute the following command:

```
:%s/\s\+$//
```

The line range "%" is used, thus this works on the whole file. The pattern that the ":substitute" command matches with is "\s\+$". This finds white space characters (\s), 1 or more of them (\+), before the end-of-line ($). Later will be explained how you write patterns like this, see usr_27.txt.
   The "to" part of the substitute command is empty: "//". Thus it replaces with nothing, effectively deleting the matched white space.

Another wasteful use of spaces is placing them before a tab. Often these can be deleted without changing the amount of white space. But not always! Therefore, you can best do this manually. Use this search command:

```
/
```

You cannot see it, but there is a space before a tab in this command. Thus it's "/<Space><Tab>".   Now use "x" to delete the space and check that the amount of white space doesn't change. You might have to insert a tab if it does change. Type "n" to find the next match. Repeat this until no more matches can be found.

12.8 Find where a word is used

If you are a Unix user, you can use a combination of Vim and the grep command to edit all the files that contain a given word. This is extremely useful if you are working on a program and want to view or edit all the files that contain a specific variable.
   For example, suppose you want to edit all the C program files that contain the word "frame_counter". To do this you use the command:

```
vim `grep -l frame_counter *.c`
```

Let's look at this command in detail. The grep command searches through a set of files for a given word. Because the -l argument is specified, the command will only list the files containing the word and not print the matching lines. The word it is searching for is "frame_counter". Actually, this can be any regular expression. (**Note:** What grep uses for regular expressions is not exactly the same as what Vim uses.)
   The entire command is enclosed in backticks (`). This tells the Unix shell to run this command and pretend that the results were typed on the command line. So what happens is that the grep command is run and produces a list of files, these files are put on the Vim command line. This results in Vim editing the file list that is the output of grep. You can then use commands like ":next" and ":first" to browse through the files.

FINDING EACH LINE

The above command only finds the files in which the word is found. You still have to find the word within the files.
   Vim has a built-in command that you can use to search a set of files for a given string. If you want to find all occurrences of "error_string" in all C program files, for example, enter the following command:

```
:grep error_string *.c
```

This causes Vim to search for the string "error_string" in all the specified files (*.c). The editor will now open the first file where a match is found and position the cursor on the first matching line. To go to the next matching line (no matter in what file it is), use the ":cnext" command. To go to the previous match, use the ":cprev" command. Use ":clist" to see all the matches and where they are.
   The ":grep" command uses the external commands grep (on Unix) or findstr (on Windows). You can change this by setting the option 'grepprg'.

Next chapter: usr_20.txt Typing command-line commands quickly

Copyright: see manual-copyright

VIM USER MANUAL   by Bram Moolenaar

Typing command-line commands quickly

Vim has a few generic features that makes it easier to enter commands. Colon commands can be abbreviated, edited and repeated. Completion is available for nearly everything.

20.1   Command line editing
20.2   Command line abbreviations
20.3   Command line completion
20.4   Command line history
20.5   Command line window

Next chapter: usr_21.txt Go away and come back
Previous chapter: usr_12.txt Clever tricks
Table of contents: usr_toc.txt

Command line editing

When you use a colon (:) command or search for a string with / or ?, Vim puts the cursor on the bottom of the screen. There you type the command or search pattern. This is called the Command line. Also when it's used for entering a search command.

The most obvious way to edit the command you type is by pressing the <BS> key. This erases the character before the cursor. To erase another character, typed earlier, first move the cursor with the cursor keys.
   For example, you have typed this:

```
:s/col/pig/
```

Before you hit <Enter>, you notice that "col" should be "cow". To correct this, you type <Left> five times. The cursor is now just after "col". Type <BS> and "w" to correct:

```
:s/cow/pig/
```

Now you can press <Enter> directly. You don't have to move the cursor to the end of the line before executing the command.

The most often used keys to move around in the command line:

<Left>	one character left
<Right>	one character right
<S-Left> or <C-Left>	one word left
<S-Right> or <C-Right>	one word right
CTRL-B or <Home>	to begin of command line
CTRL-E or <End>	to end of command line

Note:
<S-Left> (cursor left key with Shift key pressed) and <C-Left> (cursor left key with Control pressed) will not work on all keyboards. Same for the other Shift and Control combinations.

You can also use the mouse to move the cursor.

DELETING

As mentioned, <BS> deletes the character before the cursor. To delete a whole word use CTRL-W.

/the fine pig

CTRL-W

/the fine

CTRL-U removes all text, thus allows you to start all over again.

OVERSTRIKE

The <Insert> key toggles between inserting characters and replacing the existing ones. Start with this text:

/the fine pig

Move the cursor to the start of "fine" with <S-Left> twice (or <Left> eight times, if <S-Left> doesn't work). Now press <Insert> to switch to overstrike and type "great":

/the greatpig

Oops, we lost the space. Now, don't use <BS>, because it would delete the "t" (this is different from Replace mode). Instead, press <Insert> to switch from overstrike to inserting, and type the space:

/the great pig

CANCELLING

You thought of executing a : or / command, but changed your mind. To get rid of what you already typed, without executing it, press CTRL-C or <Esc>.

Note:
<Esc> is the universal "get out" key. Unfortunately, in the good old Vi pressing <Esc> in a command line executed the command! Since that might be considered to be a bug, Vim uses <Esc> to cancel the command.

But with the 'cpoptions' option it can be made Vi compatible. And when using a mapping (which might be written for Vi) <Esc> also works Vi compatible. Therefore, using CTRL-C is a method that always works.

If you are at the start of the command line, pressing <BS> will cancel the command. It's like deleting the ":" or "/" that the line starts with.

20.2 Command line abbreviations

Some of the ":" commands are really long. We already mentioned that ":substitute" can be abbreviated to ":s". This is a generic mechanism, all ":" commands can be abbreviated.

The builtin function fullcommand() can be used to return an abbreviated command's full name. For example, the following commands echo "edit", "echo", and "echomsg":

```
:echo fullcommand('e')
:echo fullcommand('ec')
:echo fullcommand('echom')
```

How short can a command get? There are 26 letters, and many more commands. For example, ":set" also starts with ":s", but ":s" doesn't start a ":set" command. Instead ":set" can be abbreviated to ":se".
When the shorter form of a command could be used for two commands, it stands for only one of them. There is no logic behind which one, you have to learn them. In the help files the shortest form that works is mentioned. For example:

```
:s[ubstitute]
```

This means that the shortest form of ":substitute" is ":s". The following characters are optional. Thus ":su" and ":sub" also work.

In the user manual we will either use the full name of command, or a short version that is still readable. For example, ":function" can be abbreviated to ":fu". But since most people don't understand what that stands for, we will use ":fun". (Vim doesn't have a ":funny" command, otherwise ":fun" would be confusing too.)

It is recommended that in Vim scripts you write the full command name. That makes it easier to read back when you make later changes. Except for some often used commands like ":w" (":write") and ":r" (":read").
A particularly confusing one is ":end", which could stand for ":endif", ":endwhile" or ":endfunction". Therefore, always use the full name.

SHORT OPTION NAMES

In the user manual the long version of the option names is used. Many options also have a short name. Unlike ":" commands, there is only one short name that works. For example, the short name of 'autoindent' is 'ai'. Thus these two commands do the same thing:

```
:set autoindent
:set ai
```

You can find the full list of long and short names here: option-list.

20.3 Command line completion

This is one of those Vim features that, by itself, is a reason to switch from Vi to Vim. Once you have used this, you can't do without.

Suppose you have a directory that contains these files:

```
info.txt
intro.txt
bodyofthepaper.txt
```

To edit the last one, you use the command:

```
:edit bodyofthepaper.txt
```

It's easy to type this wrong. A much quicker way is:

```
:edit b<Tab>
```

Which will result in the same command. What happened? The <Tab> key does completion of the word before the cursor. In this case "b". Vim looks in the directory and finds only one file that starts with a "b". That must be the one you are looking for, thus Vim completes the file name for you.

Now type:

```
:edit i<Tab>
```

Vim will beep, and give you:

```
:edit info.txt
```

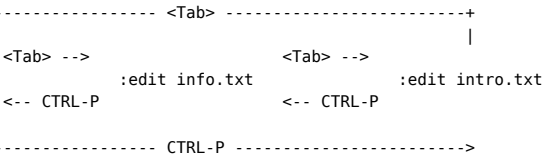
The beep means that Vim has found more than one match. It then uses the first
match it found (alphabetically). If you press <Tab> again, you get:

```
:edit intro.txt
```

Thus, if the first <Tab> doesn't give you the file you were looking for, press
it again. If there are more matches, you will see them all, one at a time.
   If you press <Tab> on the last matching entry, you will go back to what you
first typed:

```
:edit i
```

Then it starts all over again. Thus Vim cycles through the list of matches.
Use CTRL-P to go through the list in the other direction:



CONTEXT

When you type ":set i" instead of ":edit i" and press <Tab> you get:

```
:set icon
```

Hey, why didn't you get ":set info.txt"? That's because Vim has context
sensitive completion. The kind of words Vim will look for depends on the
command before it. Vim knows that you cannot use a file name just after a
":set" command, but you can use an option name.
   Again, if you repeat typing the <Tab>, Vim will cycle through all matches.
There are quite a few, it's better to type more characters first:

```
:set isk<Tab>
```

Gives:

```
:set iskeyword
```

Now type "=" and press <Tab>:

```
:set iskeyword=@,48-57,_,192-255
```

What happens here is that Vim inserts the old value of the option. Now you
can edit it.
   What is completed with <Tab> is what Vim expects in that place. Just try
it out to see how it works. In some situations you will not get what you
want. That's either because Vim doesn't know what you want, or because
completion was not implemented for that situation. In that case you will get
a <Tab> inserted (displayed as ^I).

LIST MATCHES

When there are many matches, you would like to see an overview. Do this by
pressing CTRL-D. For example, pressing CTRL-D after:

```
:set is
```

results in:

```
:set is
incsearch  isfname    isident    iskeyword
:set is
```

Vim lists the matches and then comes back with the text you typed. You can
now check the list for the item you wanted. If it isn't there, you can use
<BS> to correct the word. If there are many matches, type a few more
characters before pressing <Tab> to complete the rest.
   If you have watched carefully, you will have noticed that "incsearch"
doesn't start with "is". In this case "is" stands for the short name of
"incsearch". (Many options have a short and a long name.) Vim is clever
enough to know that you might have wanted to expand the short name of the
option into the long name.

THERE IS MORE

The CTRL-L command completes the word to the longest unambiguous string. If
you type ":edit i" and there are files "info.txt" and "info_backup.txt" you
will get ":edit info".

The 'wildmode' option can be used to change the way completion works.
The 'wildmenu' option can be used to get a menu-like list of matches.
Use the 'suffixes' option to specify files that are less important and appear
at the end of the list of files.
The 'wildignore' option specifies files that are not listed at all.

More about all of this here: cmdline-completion

20.4 Command line history

In chapter 3 we briefly mentioned the history. The basics are that you can
use the <Up> key to recall an older command line. <Down> then takes you back
to newer commands.

There are actually five histories. The ones we will mention here are for ":"
commands and for "/" and "?" search commands. The "/" and "?" commands share
the same history, because they are both search commands. The three other
histories are for expressions, debug mode commands and input lines for the
input() function. cmdline-history

Suppose you have done a ":set" command, typed ten more colon commands and then
want to repeat that ":set" command again. You could press ":" and then ten
times <Up>. There is a quicker way:

```
:se<Up>
```

Vim will now go back to the previous command that started with "se". You have
a good chance that this is the ":set" command you were looking for. At least
you should not have to press <Up> very often (unless ":set" commands is all
you have done).

The <Up> key will use the text typed so far and compare it with the lines in
the history. Only matching lines will be used.
   If you do not find the line you were looking for, use <Down> to go back to
what you typed and correct that. Or use CTRL-U to start all over again.

To see all the lines in the history:

```
:history
```

That's the history of ":" commands. The search history is displayed with this
command:

```
:history /
```

CTRL-P will work like <Up>, except that it doesn't matter what you already
typed. Similarly for CTRL-N and <Down>. CTRL-P stands for previous, CTRL-N
for next.

20.5 Command line window

Typing the text in the command line works differently from typing text in
Insert mode. It doesn't allow many commands to change the text. For most
commands that's OK, but sometimes you have to type a complicated command.
That's where the command line window is useful.

Open the command line window with this command:

```
q:
```

Vim now opens a (small) window at the bottom. It contains the command line
history, and an empty line at the end:

```
+-----------------------------------+
|other window                       |
|~                                  |
|file.txt===========================|
|:e c                               |
|:e config.h.in                     |
|:set path=.,/usr/include,,         |
|:set iskeyword=@,48-57,_,192-255   |
|:set is                            |
|:q                                 |
|:                                  |
|command-line======================|
|                                   |
+-----------------------------------+
```

You are now in Normal mode. You can use the "hjkl" keys to move around. For
example, move up with "5k" to the ":e config.h.in" line. Type "$h" to go to
the "i" of "in" and type "cwout". Now you have changed the line to:

:e config.h.out

Now press <Enter> and this command will be executed. The command line window
will close.
   The <Enter> command will execute the line under the cursor. It doesn't
matter whether Vim is in Insert mode or in Normal mode.
   Changes in the command line window are lost. They do not result in the

history to be changed. Except that the command you execute will be added to the end of the history, like with all executed commands.

The command line window is very useful when you want to have overview of the history, lookup a similar command, change it a bit and execute it. A search command can be used to find something.

In the previous example the "?config" search command could have been used to find the previous command that contains "config". It's a bit strange, because you are using a command line to search in the command line window. While typing that search command you can't open another command line window, there can be only one.

Next chapter: usr_21.txt Go away and come back

Copyright: see manual-copyright

VIM USER MANUAL by Bram Moolenaar

Go away and come back

This chapter goes into mixing the use of other programs with Vim. Either by executing program from inside Vim or by leaving Vim and coming back later. Furthermore, this is about the ways to remember the state of Vim and restore it later.

21.1 Suspend and resume
21.2 Executing shell commands
21.3 Remembering information; ShaDa
21.4 Sessions
21.5 Views
21.6 Modelines

Next chapter: usr_22.txt Finding the file to edit
Previous chapter: usr_20.txt Typing command-line commands quickly
Table of contents: usr_toc.txt

Suspend and resume

Like most Unix programs Vim can be suspended by pressing CTRL-Z. This stops Vim and takes you back to the shell it was started in. You can then do any other commands until you are bored with them. Then bring back Vim with the "fg" command.

```
CTRL-Z
{any sequence of shell commands}
fg
```

You are right back where you left Vim, nothing has changed.
In case pressing CTRL-Z doesn't work, you can also use ":suspend". Don't forget to bring Vim back to the foreground, you would lose any changes that you made!

Only Unix has support for this. On other systems Vim will start a shell for you. This also has the functionality of being able to execute shell commands. But it's a new shell, not the one that you started Vim from.
When you are running the GUI you can't go back to the shell where Vim was started. CTRL-Z will minimize the Vim window instead.

21.2 Executing shell commands

To execute a single shell command from Vim use ":!{command}". For example, to see a directory listing:

```
:!ls
:!dir
```

The first one is for Unix, the second one for MS-Windows.
Vim will execute the program. When it ends you will get a prompt to hit <Enter>. This allows you to have a look at the output from the command before returning to the text you were editing.
The "!" is also used in other places where a program is run. Let's take a look at an overview:

```
:!{program}            execute {program}
:r !{program}          execute {program} and read its output
:w !{program}          execute {program} and send text to its input
:[range]!{program}     filter text through {program}
```

Notice that the presence of a range before "!{program}" makes a big difference. Without it executes the program normally, with the range a number of text lines is filtered through the program.

Executing a whole row of programs this way is possible. But a shell is much better at it. You can start a new shell with :terminal.

This is similar to using CTRL-Z to suspend Vim. The difference is that a new shell is started.

21.3 Remembering information; ShaDa

After editing for a while you will have text in registers, marks in various files, a command line history filled with carefully crafted commands. When you exit Vim all of this is lost. But you can get it back!

The ShaDa (abbreviation of SHAred DAta) file is designed to store status information:

```
Command-line and Search pattern history
Text in registers
Marks for various files
The buffer list
Global variables
```

Each time you exit Vim it will store this information in a file, the ShaDa file. When Vim starts again, the ShaDa file is read and the information restored.

The 'shada' option is set by default to restore a limited number of items. You might want to set it to remember more information. This is done through the following command:

```
:set shada=string
```

The string specifies what to save. The syntax of this string is an option
character followed by an argument. The option/argument pairs are separated by
commas.
   Take a look at how you can build up your own shada string. First, the '
option is used to specify how many files for which you save marks (a-z). Pick
a nice even number for this option (1000, for instance). Your command now
looks like this:

```
:set shada='1000
```

The f option controls whether global marks (A-Z and 0-9) are stored. If this
option is 0, none are stored. If it is 1 or you do not specify an f option,
the marks are stored. You want this feature, so now you have this:

```
:set shada='1000,f1
```

The < option controls how many lines are saved for each of the registers. By
default, all the lines are saved. If 0, nothing is saved. To avoid adding
thousands of lines to your ShaDa file (which might never get used and makes
starting Vim slower) you use a maximum of 500 lines:

```
:set shada='1000,f1,<500
```

Other options you might want to use:
```
        :       number of lines to save from the command line history
        @       number of lines to save from the input line history
        /       number of lines to save from the search history
        r       removable media, for which no marks will be stored (can be
                used several times)
        !       global variables that start with an uppercase letter and
                don't contain lowercase letters
        h       disable 'hlsearch' highlighting when starting
        %       the buffer list (only restored when starting Vim without file
                arguments)
        c       convert the text using 'encoding'
        n       name used for the ShaDa file (must be the last option)
```

See the 'shada' option and shada-file for more information.

When you run Vim multiple times, the last one exiting will store its
information. This may cause information that previously exiting Vims stored
to be lost. Each item can be remembered only once.

GETTING BACK TO WHERE YOU STOPPED VIM

You are halfway through editing a file and it's time to leave for holidays.
You exit Vim and go enjoy yourselves, forgetting all about your work. After a
couple of weeks you start Vim, and type:

```
'0
```

And you are right back where you left Vim. So you can get on with your work.
   Vim creates a mark each time you exit Vim. The last one is '0. The
position that '0 pointed to is made '1. And '1 is made to '2, and so forth.
Mark '9 is lost.
   The :marks command is useful to find out where '0 to '9 will take you.

GETTING BACK TO SOME FILE

If you want to go back to a file that you edited recently, but not when
exiting Vim, there is a slightly more complicated way. You can see a list of
files by typing the command:

```
:oldfiles
```

1: ~/.config/nvim/init.vim
2: ~/text/resume.txt
3: /tmp/draft

Now you would like to edit the second file, which is in the list preceded by
"2:". You type:

```
:e #<2
```

Instead of ":e" you can use any command that has a file name argument, the
"#<2" item works in the same place as "%" (current file name) and "#"
(alternate file name). So you can also split the window to edit the third
file:

```
:split #<3
```

That #<123 thing is a bit complicated when you just want to edit a file.
Fortunately there is a simpler way:

```
:browse oldfiles
```

1: ~/.config/nvim/init.vim
2: ~/text/resume.txt
3: /tmp/draft
        -- More --

You get the same list of files as with :oldfiles. If you want to edit
"resume.txt" first press "q" to stop the listing. You will get a prompt:

Type number and <Enter> (empty cancels):

Type "2" and press <Enter> to edit the second file.

More info at :oldfiles, v:oldfiles and c_#<.

MOVE INFO FROM ONE VIM TO ANOTHER

You can use the ":wshada" and ":rshada" commands to save and restore the
information while still running Vim. This is useful for exchanging register
contents between two instances of Vim, for example. In the first Vim do:

```
:wshada! ~/tmp/shada
```

And in the second Vim do:

```
:rshada! ~/tmp/shada
```

Obviously, the "w" stands for "write" and the "r" for "read".
   The ! character is used by ":wshada" to forcefully overwrite an existing
file. When it is omitted, and the file exists, the information is merged into
the file.
   The ! character used for ":rshada" means that all the information in ShaDa
file has priority over existing information, this may overwrite it. Without
the ! only information that wasn't set is used.
   These commands can also be used to store info and use it again later. You
could make a directory full of ShaDa files, each containing info for a
different purpose.

21.4 Sessions

Suppose you are editing along, and it is the end of the day. You want to quit
work and pick up where you left off the next day. You can do this by saving
your editing session and restoring it the next day.
   A Vim session contains all the information about what you are editing.
This includes things such as the file list, window layout, global variables,
options and other information. (Exactly what is remembered is controlled by
the 'sessionoptions' option, described below.)
   The following command creates a session file:

```
:mksession vimbook.vim
```

Later if you want to restore this session, you can use this command:

```
:source vimbook.vim
```

If you want to start Vim and restore a specific session, you can use the
following command:

```
vim -S vimbook.vim
```

This tells Vim to read a specific file on startup. The 'S' stands for
session (actually, you can source any Vim script with -S, thus it might as
well stand for "source").

The windows that were open are restored, with the same position and size as
before. Mappings and option values are like before.
   What exactly is restored depends on the 'sessionoptions' option. The
default value is:
"blank,buffers,curdir,folds,help,options,tabpages,winsize,terminal".

```
        blank           keep empty windows
        buffers         all buffers, not only the ones in a window
        curdir          the current directory
        folds           folds, also manually created ones
        help            the help window
        options         all options and mappings
        tabpages        all tab pages
        winsize         window sizes
        terminal        include terminal windows
```

Change this to your liking. To also restore the size of the Vim window, for
example, use:

```
:set sessionoptions+=resize
```

SESSION HERE, SESSION THERE

The obvious way to use sessions is when working on different projects.
Suppose you store your session files in the directory "~/.config/nvim". You
are currently working on the "secret" project and have to switch to the
"boring" project:

```
:wall
:mksession! ~/.config/nvim/secret.vim
:source ~/.config/nvim/boring.vim
```

This first uses ":wall" to write all modified files. Then the current session
is saved, using ":mksession!". This overwrites the previous session. The
next time you load the secret session you can continue where you were at this
point. And finally you load the new "boring" session.

If you open help windows, split and close various windows, and generally mess
up the window layout, you can go back to the last saved session:

```
:source ~/.config/nvim/boring.vim
```

Thus you have complete control over whether you want to continue next time
where you are now, by saving the current setup in a session, or keep the
session file as a starting point.
   Another way of using sessions is to create a window layout that you like to
use, and save this in a session. Then you can go back to this layout whenever
you want.
   For example, this is a nice layout to use:

```
+----------------------------------------+
|                  VIM - main help file  |
|                                        |
|Move around:  Use the cursor keys, or "h|
|help.txt================================|
|explorer   |                            |
|dir        |~                           |
|dir        |~                           |
|file       |~                           |
|file       |~                           |
|file       |~                           |
|file       |~                           |
|~/=========|[No File]===================|
|                                        |
+----------------------------------------+
```

This has a help window at the top, so that you can read this text. The narrow
vertical window on the left contains a file explorer. This is a Vim plugin
that lists the contents of a directory. You can select files to edit there.
More about this in the next chapter.
   Create this from a just started Vim with:

```
:help
CTRL-W w
:vertical split ~/
```

You can resize the windows a bit to your liking. Then save the session with:

```
:mksession ~/.config/nvim/mine.vim
```

Now you can start Vim with this layout:

```
vim -S ~/.config/nvim/mine.vim
```

Hint: To open a file you see listed in the explorer window in the empty
window, move the cursor to the filename and press "O". Double clicking with
the mouse will also do this.

SESSIONS AND SHADA

Sessions store many things, but not the position of marks, contents of
registers and the command line history. You need to use the shada feature
for these things.
   In most situations you will want to use sessions separately from shada.
This can be used to switch to another session, but keep the command line
history. And yank text into registers in one session, and paste it back in
another session.
   You might prefer to keep the info with the session. You will have to do
this yourself then. Example:

```
:mksession! ~/.config/nvim/secret.vim
:wshada! ~/.local/state/nvim/shada/secret.shada
```

And to restore this again:

```
:source ~/.config/nvim/secret.vim
:rshada! ~/.local/state/nvim/shada/secret.shada
```

21.5 Views

A session stores the looks of the whole of Vim. When you want to store the properties for one window only, use a view.
   The use of a view is for when you want to edit a file in a specific way. For example, you have line numbers enabled with the 'number' option and defined a few folds. Just like with sessions, you can remember this view on the file and restore it later. Actually, when you store a session, it stores the view of each window.
   There are two basic ways to use views. The first is to let Vim pick a name for the view file. You can restore the view when you later edit the same file. To store the view for the current window:

```
:mkview
```

Vim will decide where to store the view. When you later edit the same file you get the view back with this command:

```
:loadview
```

That's easy, isn't it?
   Now you want to view the file without the 'number' option on, or with all folds open, you can set the options to make the window look that way. Then store this view with:

```
:mkview 1
```

Obviously, you can get this back with:

```
:loadview 1
```

Now you can switch between the two views on the file by using ":loadview" with and without the "1" argument.
   You can store up to ten views for the same file this way, one unnumbered and nine numbered 1 to 9.

A VIEW WITH A NAME

The second basic way to use views is by storing the view in a file with a name you choose. This view can be loaded while editing another file. Vim will then switch to editing the file specified in the view. Thus you can use this to quickly switch to editing another file, with all its options set as you saved them.
   For example, to save the view of the current file:

```
:mkview ~/.config/nvim/main.vim
```

You can restore it with:

```
:source ~/.config/nvim/main.vim
```

21.6 Modelines

When editing a specific file, you might set options specifically for that file. Typing these commands each time is boring. Using a session or view for editing a file doesn't work when sharing the file between several people.
   The solution for this situation is adding a modeline to the file. This is a line of text that tells Vim the values of options, to be used in this file only.
   A typical example is a C program where you make indents by a multiple of 4 spaces. This requires setting the 'shiftwidth' option to 4. This modeline will do that:

```
/* vim:set shiftwidth=4: */ ~
```

Put this line as one of the first or last five lines in the file. When editing the file, you will notice that 'shiftwidth' will have been set to four. When editing another file, it's set back to the default value of eight.
   For some files the modeline fits well in the header, thus it can be put at the top of the file. For text files and other files where the modeline gets in the way of the normal contents, put it at the end of the file.

The 'modelines' option specifies how many lines at the start and end of the file are inspected for containing a modeline. To inspect ten lines:

```
:set modelines=10
```

The 'modeline' option can be used to switch this off. Do this when you are working as root on Unix or Administrator on MS-Windows, or when you don't trust the files you are editing:

```
:set nomodeline
```

Use this format for the modeline:

```
any-text vim:set {option}={value} ... : any-text
```

The "any-text" indicates that you can put any text before and after the part
that Vim will use. This allows making it look like a comment, like what was
done above with "/*" and "*/".
   The " vim:" part is what makes Vim recognize this line. There must be
white space before "vim", or "vim" must be at the start of the line. Thus
using something like "gvim:" will not work.
   The part between the colons is a ":set" command. It works the same way as
typing the ":set" command, except that you need to insert a backslash before a
colon (otherwise it would be seen as the end of the modeline).

Another example:

```
// vim:set textwidth=72 dir=c\:\tmp:  use c:\tmp here
```

There is an extra backslash before the first colon, so that it's included in
the ":set" command. The text after the second colon is ignored, thus a remark
can be placed there.

For more details see modeline.

Next chapter: usr_22.txt Finding the file to edit

Copyright: see manual-copyright

Finding the file to edit

Files can be found everywhere. So how do you find them? Vim offers various ways to browse the directory tree. There are commands to jump to a file that is mentioned in another. And Vim remembers which files have been edited before.

22.1 The file browser
22.2 The current directory
22.3 Finding a file
22.4 The buffer list

Next chapter: usr_23.txt Editing other files
Previous chapter: usr_21.txt Go away and come back
Table of contents: usr_toc.txt

The file browser

Nvim can edit a directory. Try this:

```
:edit .
```

The window will be filled with a directory listing. It looks like this:

```
check/
autoload.txt
change.txt
eval.txt
filetype.txt
help.txt
```

Directories end in "/".

You can use Normal mode Vim commands to move around in the text. For example, move the cursor atop a file and press <Enter>; you will then be editing that file. Press "-" to open the parent directory of the current file or directory. CTRL-O also works.

Try using <Enter> while the cursor is atop a directory name. The result is that the file browser moves into that directory and displays the items found there. Pressing "-" moves you one level higher, to the parent directory. "R" reloads the listing.

The directory listing opened by ":edit ." is a viewer, not a file manager. For netrw's browser, file operations, and remote directories, use ":Explore":

```
:Explore .
```

The netrw browser may also use URLs such as:

```
:Explore ftp://somehost/path/to/dir/
:e scp://somehost/path/to/dir/
```

Note: The trailing "/" is important.

See netrw-browse for more.

22.2 The current directory

Just like the shell, Vim has the concept of a current directory. Suppose you are in your home directory and want to edit several files in a directory "VeryLongFileName". You could do:

```
:edit VeryLongFileName/file1.txt
:edit VeryLongFileName/file2.txt
:edit VeryLongFileName/file3.txt
```

To avoid much of the typing, do this:

```
:cd VeryLongFileName
:edit file1.txt
:edit file2.txt
:edit file3.txt
```

The ":cd" command changes the current directory. You can see what the current directory is with the ":pwd" command:

```
:pwd
/home/Bram/VeryLongFileName
```

Vim remembers the last directory that you used. Use "cd -" to go back to it.
Example:

```
:pwd
/home/Bram/VeryLongFileName
:cd /etc
:pwd
/etc
:cd -
:pwd
/home/Bram/VeryLongFileName
:cd -
:pwd
/etc
```

WINDOW LOCAL DIRECTORY

When you split a window, both windows use the same current directory. When you want to edit a number of files somewhere else in the new window, you can make it use a different directory, without changing the current directory in the other window. This is called a local directory.

```
:pwd
/home/Bram/VeryLongFileName
:split
:lcd /etc
:pwd
/etc
CTRL-W w
:pwd
/home/Bram/VeryLongFileName
```

So long as no :lcd command has been used, all windows share the same current directory. Doing a :cd command in one window will also change the current directory of the other window.

For a window where :lcd has been used a different current directory is remembered. Using :cd or :lcd in other windows will not change it.

When using a :cd command in a window that uses a different current directory, it will go back to using the shared directory.

TAB LOCAL DIRECTORY

When you open a new tabpage, it uses the directory of the window in the previous tabpage from which the new tabpage was opened. You can change the directory of the current tabpage using the :tcd command. All the windows in a tabpage share this directory except for windows with a window-local directory. Any new windows opened in this tabpage will use this directory as the current working directory. Using a :cd command in a tabpage will not change the working directory of tabpages which have a tab local directory. When the global working directory is changed using the :cd command in a tab page, it will also change the current tabpage working directory.

22.3 Finding a file

You are editing a C program that contains this line:

#include "inits.h"

You want to see what is in that "inits.h" file. Move the cursor on the name of the file and type:

```
gf
```

Vim will find the file and edit it.

What if the file is not in the current directory? Vim will use the 'path' option to find the file. This option is a list of directory names where to look for your file.

Suppose you have your include files located in "c:/prog/include". This command will add it to the 'path' option:

```
:set path+=c:/prog/include
```

This directory is an absolute path. No matter where you are, it will be the same place. What if you have located files in a subdirectory, below where the file is? Then you can specify a relative path name. This starts with a dot:

```
:set path+=./proto
```

This tells Vim to look in the directory "proto", below the directory where the file in which you use "gf" is. Thus using "gf" on "inits.h" will make Vim look for "proto/inits.h", starting in the directory of the file.

Without the "./", thus "proto", Vim would look in the "proto" directory below the current directory. And the current directory might not be where the file that you are editing is located.

The 'path' option allows specifying the directories where to search for files in many more ways. See the help on the 'path' option.

The 'isfname' option is used to decide which characters are included in the

file name, and which ones are not (e.g., the " character in the example above).

When you know the file name, but it's not to be found in the file, you can type it:

```
:find inits.h
```

Vim will then use the 'path' option to try and locate the file. This is the same as the ":edit" command, except for the use of 'path'.

To open the found file in a new window use CTRL-W f instead of "gf", or use ":sfind" instead of ":find".

A nice way to directly start Vim to edit a file somewhere in the 'path':

```
vim "+find stdio.h"
```

This finds the file "stdio.h" in your value of 'path'. The quotes are necessary to have one argument -+c.

22.4 The buffer list

The Vim editor uses the term buffer to describe a file being edited. Actually, a buffer is a copy of the file that you edit. When you finish changing the buffer, you write the contents of the buffer to the file. Buffers not only contain file contents, but also all the marks, settings, and other stuff that goes with it.

HIDDEN BUFFERS

Suppose you are editing the file one.txt and need to edit the file two.txt. You could simply use ":edit two.txt", but since you made changes to one.txt that won't work. You also don't want to write one.txt yet. Vim has a solution for you:

```
:hide edit two.txt
```

The buffer "one.txt" disappears from the screen, but Vim still knows that you are editing this buffer, so it keeps the modified text. This is called a hidden buffer: The buffer contains text, but you can't see it.
   The argument of ":hide" is another command. ":hide" makes that command behave as if the 'hidden' option was set. You could also set this option yourself. The effect is that when any buffer is abandoned, it becomes hidden.
   Be careful! When you have hidden buffers with changes, don't exit Vim without making sure you have saved all the buffers.

INACTIVE BUFFERS

When a buffer has been used once, Vim remembers some information about it. When it is not displayed in a window and it is not hidden, it is still in the buffer list. This is called an inactive buffer. Overview:

```
        Active          Appears in a window, text loaded.
        Hidden          Not in a window, text loaded.
        Inactive        Not in a window, no text loaded.
```

The inactive buffers are remembered, because Vim keeps information about them, like marks. And remembering the file name is useful too, so that you can see which files you have edited. And edit them again.

LISTING BUFFERS

View the buffer list with this command:

```
:buffers
```

A command which does the same, is not so obvious to list buffers, but is much shorter to type:

```
:ls
```

The output could look like this:

```
  1 #h "help.txt"                   line 62
  2 %a + "usr_21.txt" line 1
  3 "usr_toc.txt" line 1
```

The first column contains the buffer number. You can use this to edit the buffer without having to type the name, see below.
   After the buffer number come the flags. Then the name of the file and the line number where the cursor was the last time.
   The flags that can appear are these (from left to right):

```
        u       Buffer is unlisted unlisted-buffer.
         %      Current buffer.
         #      Alternate buffer.
          a     Buffer is loaded and displayed.
```

```
          h     Buffer is loaded but hidden.
           =    Buffer is read-only.
           -    Buffer is not modifiable, the 'modifiable' option is off.
            +   Buffer has been modified.
```

EDITING A BUFFER

You can edit a buffer by its number. That avoids having to type the file
name:

```
:buffer 2
```

But the only way to know the number is by looking in the buffer list. You can
use the name, or part of it, instead:

```
:buffer help
```

Vim will find the best match for the name you type. If there is only one
buffer that matches the name, it will be used. In this case "help.txt".
   To open a buffer in a new window:

```
:sbuffer 3
```

This works with a name as well.

USING THE BUFFER LIST

You can move around in the buffer list with these commands:

```
        :bnext          go to next buffer
        :bprevious      go to previous buffer
        :bfirst         go to the first buffer
        :blast          go to the last buffer
```

To remove a buffer from the list, use this command:

```
:bdelete 3
```

Again, this also works with a name.
   If you delete a buffer that was active (visible in a window), that window
will be closed. If you delete the current buffer, the current window will be
closed. If it was the last window, Vim will find another buffer to edit. You
can't be editing nothing!

> **Note:** Even after removing the buffer with ":bdelete" Vim still
> remembers it. It's actually made "unlisted", it no longer appears in
> the list from ":buffers". The ":buffers!" command will list unlisted
> buffers (yes, Vim can do the impossible). To really make Vim forget
> about a buffer, use ":bwipe". Also see the 'buflisted' option.

Next chapter: usr_23.txt Editing other files

Copyright: see manual-copyright

Editing other files

This chapter is about editing files that are not ordinary files. With Vim you can edit files that are compressed. Some files need to be accessed over the internet. With some restrictions, binary files can be edited as well.

23.1 DOS, Mac and Unix files
23.2 Files on the internet
23.3 Binary files
23.4 Compressed files

Next chapter: usr_24.txt Inserting quickly
Previous chapter: usr_22.txt Finding the file to edit
Table of contents: usr_toc.txt

DOS, Mac and Unix files

Back in the early days, the old Teletype machines used two characters to start a new line. One to move the carriage back to the first position (carriage return, <CR>), another to move the paper up (line feed, <LF>).

When computers came out, storage was expensive. Some people decided that they did not need two characters for end-of-line. The Unix people decided they could use <New Line> or <NL> only for end-of-line. The Apple people standardized on <CR>. The Microsoft Windows folks decided to keep the old <CR><NL> (we use <NL> for line feed in the help text).

This means that if you try to move a file from one system to another, you have line-break problems. The Vim editor automatically recognizes the different file formats and handles things properly behind your back.

The option 'fileformats' contains the various formats that will be tried when a new file is edited. The following command, for example, tells Vim to try Unix format first and MS-DOS format second:

```
:set fileformats=unix,dos
```

You will notice the format in the message you get when editing a file. You don't see anything if you edit a native file format. Thus editing a Unix file on Unix won't result in a remark. But when you edit a dos file, Vim will notify you of this:

"/tmp/test" [dos] 3L, 71C

For a Mac file you would see "[mac]".

The detected file format is stored in the 'fileformat' option. To see which format you have, execute the following command:

```
:set fileformat?
```

The three names that Vim uses are:

unix	<NL>
dos	<CR><NL>
mac	<CR>

USING THE MAC FORMAT

On Unix, <NL> is used to break a line. It's not unusual to have a <CR> character halfway in a line. Incidentally, this happens quite often in Vi (and Vim) scripts.

On the Macintosh, where <CR> is the line break character, it's possible to have a <NL> character halfway in a line.

The result is that it's not possible to be 100% sure whether a file containing both <CR> and <NL> characters is a Mac or a Unix file. Therefore, Vim assumes that on Unix you probably won't edit a Mac file, and doesn't check for this type of file. To check for this format anyway, add "mac" to 'fileformats':

```
:set fileformats+=mac
```

Then Vim will take a guess at the file format. Watch out for situations where Vim guesses wrong.

OVERRULING THE FORMAT

If you use the good old Vi and try to edit an MS-DOS format file, you will find that each line ends with a ^M character. (^M is <CR>). The automatic detection avoids this. Suppose you do want to edit the file that way? Then you need to overrule the format:

```
:edit ++ff=unix file.txt
```

The "++" string is an item that tells Vim that an option name follows, which overrules the default for this single command. "++ff" is used for 'fileformat'. You could also use "++ff=mac" or "++ff=dos".

This doesn't work for any option, only "++ff" and "++enc" are currently implemented. The full names "++fileformat" and "++encoding" also work.

CONVERSION

You can use the 'fileformat' option to convert from one file format to another. Suppose, for example, that you have an MS-DOS file named README.TXT that you want to convert to Unix format. Start by editing the MS-DOS format file:

```
vim README.TXT
```

Vim will recognize this as a dos format file. Now change the file format to Unix:

```
:set fileformat=unix
:write
```

The file is written in Unix format.

23.2 Files on the internet

Someone sends you an e-mail message, which refers to a file by its URL. For example:

You can find the information here:
https://ftp.nluug.nl/pub/vim/README

You could start a program to download the file, save it on your local disk and then start Vim to edit it.
   There is a much simpler way. Move the cursor to any character of the URL. Then use this command:

```
gf
```

With a bit of luck, Vim will figure out which program to use for downloading the file, download it and edit the copy. To open the file in a new window use CTRL-W f.
   If something goes wrong you will get an error message. It's possible that the URL is wrong, you don't have permission to read it, the network connection is down, etc. Unfortunately, it's hard to tell the cause of the error. You might want to try the manual way of downloading the file.

Accessing files over the internet works with the netrw plugin. Currently URLs with these formats are recognized:

```
        ftp://          uses ftp
        rcp://          uses rcp
        scp://          uses scp
        http://         uses wget (reading only)
```

Vim doesn't do the communication itself, it relies on the mentioned programs to be available on your computer. On most Unix systems "ftp" and "rcp" will be present. "scp" and "wget" might need to be installed.

Vim detects these URLs for each command that starts editing a new file, also with ":edit" and ":split", for example. Write commands also work, except for http://.

For more information, also about passwords, see netrw.

23.3 Binary files

You can edit binary files with Vim. Vim wasn't really made for this, thus there are a few restrictions. But you can read a file, change a character and write it back, with the result that only that one character was changed and the file is identical otherwise.
   To make sure that Vim does not use its clever tricks in the wrong way, add the "-b" argument when starting Vim:

```
vim -b datafile
```

This sets the 'binary' option. The effect of this is that unexpected side effects are turned off. For example, 'textwidth' is set to zero, to avoid automatic formatting of lines. And files are always read in Unix file format.

Binary mode can be used to change a message in a program. Be careful not to insert or delete any characters, it would stop the program from working. Use "R" to enter replace mode.

Many characters in the file will be unprintable. To see them in Hex format:

```
:set display=uhex
```

Otherwise, the "ga" command can be used to see the value of the character under the cursor. The output, when the cursor is on an <Esc>, looks like this:

<^[> 27, Hex 1b, Octal 033

There might not be many line breaks in the file. To get some overview switch the 'wrap' option off:

```
:set nowrap
```

BYTE POSITION

To see on which byte you are in the file use this command:

```
g CTRL-G
```

The output is verbose:

Col 9-16 of 9-16; Line 277 of 330; Word 1806 of 2058; Byte 10580 of 12206

The last two numbers are the byte position in the file and the total number of bytes. This takes into account how 'fileformat' changes the number of bytes that a line break uses.
    To move to a specific byte in the file, use the "go" command. For example, to move to byte 2345:

```
2345go
```

USING XXD

A real binary editor shows the text in two ways: as it is and in hex format. You can do this in Vim by first converting the file with the "xxd" program. This comes with Vim.
   First edit the file in binary mode:

```
vim -b datafile
```

Now convert the file to a hex dump with xxd:

```
:%!xxd
```

The text will look like this:

```
0000000: 1f8b 0808 39d7 173b 0203 7474 002b 4e49  ....9..;..tt.+NI
0000010: 4b2c 8660 eb9c ecac c462 eb94 345e 2e30  K,.`.....b..4^.0
0000020: 373b 2731 0b22 0ca6 c1a2 d669 1035 39d9  7;'1."......i.59.
```

You can now view and edit the text as you like. Vim treats the information as ordinary text. Changing the hex does not cause the printable character to be changed, or the other way around.
   Finally convert it back with:

```
:%!xxd -r
```

Only changes in the hex part are used. Changes in the printable text part on the right are ignored.

See the manual page of xxd for more information.

23.4 Compressed files

This is easy: You can edit a compressed file just like any other file. The "gzip" plugin takes care of decompressing the file when you edit it. And compressing it again when you write it.
   These compression methods are currently supported:

```
       .Z      compress
       .gz     gzip
       .bz2    bzip2
```

Vim uses the mentioned programs to do the actual compression and decompression. You might need to install the programs first.

Next chapter: usr_24.txt Inserting quickly

Copyright: see manual-copyright

Inserting quickly

When entering text, Vim offers various ways to reduce the number of keystrokes and avoid typing mistakes. Use Insert mode completion to repeat previously typed words. Abbreviate long words to short ones. Type characters that aren't on your keyboard.

24.1 Making corrections
24.2 Showing matches
24.3 Completion
24.4 Repeating an insert
24.5 Copying from another line
24.6 Inserting a register
24.7 Abbreviations
24.8 Entering special characters
24.9 Digraphs
24.10 Normal mode commands

Next chapter: usr_25.txt Editing formatted text
Previous chapter: usr_23.txt Editing other files
Table of contents: usr_toc.txt

Making corrections

The <BS> key was already mentioned. It deletes the character just before the cursor. The <Del> key does the same for the character under (after) the cursor.

When you typed a whole word wrong, use CTRL-W:

The horse had fallen to the sky

CTRL-W

The horse had fallen to the

If you really messed up a line and want to start over, use CTRL-U to delete it. This keeps the text after the cursor and the indent. Only the text from the first non-blank to the cursor is deleted. With the cursor on the "f" of "fallen" in the next line pressing CTRL-U does this:

The horse had fallen to the

CTRL-U

fallen to the

When you spot a mistake a few words back, you need to move the cursor there to correct it. For example, you typed this:

The horse had follen to the ground

You need to change "follen" to "fallen". With the cursor at the end, you would type this to correct it:

```
<Esc>4blraA
```

```
       get out of Insert mode          <Esc>
       four words back                      4b
       move on top of the "o"                 l
       replace with "a"                        ra
       restart Insert mode                       A
```

Another way to do this:

```
<C-Left><C-Left><C-Left><C-Left><Right><Del>a<End>
```

```
       four words back              <C-Left><C-Left><C-Left><C-Left>
       move on top of the "o"                   <Right>
       delete the "o"                                <Del>
       insert an "a"                                      a
       go to end of the line                               <End>
```

This uses special keys to move around, while remaining in Insert mode. This resembles what you would do in a modeless editor. It's easier to remember, but takes more time (you have to move your hand from the letters to the cursor keys, and the <End> key is hard to press without looking at the keyboard).

These special keys are most useful when writing a mapping that doesn't leave Insert mode. The extra typing doesn't matter then.

An overview of the keys you can use in Insert mode:

```
       <C-Home>        to start of the file
       <PageUp>        a whole screenful up
       <Home>          to start of line
       <S-Left>        one word left
       <C-Left>        one word left
       <S-Right>       one word right
       <C-Right>       one word right
       <End>           to end of the line
       <PageDown>      a whole screenful down
       <C-End>         to end of the file
```

There are a few more, see ins-special-special.

24.2 Showing matches

When you type a) it would be nice to see with which (it matches. To make Vim do that use this command:

```
:set showmatch
```

When you now type a text like "(example)", as soon as you type the) Vim will briefly move the cursor to the matching (, keep it there for half a second, and move back to where you were typing.
   In case there is no matching (, Vim will beep. Then you know that you might have forgotten the (somewhere, or typed a) too many.
   The match will also be shown for [] and {} pairs. You don't have to wait with typing the next character, as soon as Vim sees it the cursor will move back and inserting continues as before.
   You can change the time Vim waits with the 'matchtime' option. For example, to make Vim wait one and a half second:

```
:set matchtime=15
```

The time is specified in tenths of a second.

24.3 Completion

Vim can automatically complete words on insertion. You type the first part of a word, press CTRL-P, and Vim guesses the rest.
   Suppose, for example, that you are creating a C program and want to type in the following:

total = ch_array[0] + ch_array[1] + ch_array[2];

You start by entering the following:

total = ch_array[0] + ch_

At this point, you tell Vim to complete the word using the command CTRL-P. Vim searches for a word that starts with what's in front of the cursor. In this case, it is "ch_", which matches with the word ch_array. So typing CTRL-P gives you the following:

total = ch_array[0] + ch_array

After a little more typing, you get this (ending in a space):

total = ch_array[0] + ch_array[1] +

If you now type CTRL-P Vim will search again for a word that completes the word before the cursor. Since there is nothing in front of the cursor, it finds the first word backwards, which is "ch_array". Typing CTRL-P again gives you the next word that matches, in this case "total". A third CTRL-P searches further back. If there is nothing else, it causes the editor to run out of words, so it returns to the original text, which is nothing. A fourth CTRL-P causes the editor to start over again with "ch_array".

To search forward, use CTRL-N. Since the search wraps around the end of the file, CTRL-N and CTRL-P will find the same matches, but in a different sequence. Hint: CTRL-N is Next-match and CTRL-P is Previous-match.

The Vim editor goes through a lot of effort to find words to complete. By default, it searches the following places:

1. Current file
2. Files in other windows
3. Other loaded files (hidden buffers)
4. Files which are not loaded (inactive buffers)
5. Tag files
6. All files #included by the current file

OPTIONS

You can customize the search order with the 'complete' option.

The 'ignorecase' option is used. When it is set, case differences are ignored when searching for matches.

A special option for completion is 'infercase'. This is useful to find matches while ignoring case ('ignorecase' must be set) but still using the case of the word typed so far. Thus if you type "For" and Vim finds a match "fortunately", it will result in "Fortunately".

COMPLETING SPECIFIC ITEMS

If you know what you are looking for, you can use these commands to complete with a certain type of item:

```
        CTRL-X CTRL-F           file names
        CTRL-X CTRL-L           whole lines
        CTRL-X CTRL-D           macro definitions (also in included files)
```

```
        CTRL-X CTRL-I           current and included files
        CTRL-X CTRL-K           words from a dictionary
        CTRL-X CTRL-R           contents from registers
        CTRL-X CTRL-T           words from a thesaurus
        CTRL-X CTRL-]           tags
        CTRL-X CTRL-V           Vim command line
```

After each of them CTRL-N can be used to find the next match, CTRL-P to find the previous match.
   More information for each of these commands here: ins-completion.

COMPLETING FILE NAMES

Let's take CTRL-X CTRL-F as an example. This will find file names. It scans the current directory for files and displays each one that matches the word in front of the cursor.
   Suppose, for example, that you have the following files in the current directory:

```
        main.c  sub_count.c  sub_done.c  sub_exit.c
```

Now enter Insert mode and start typing:

The exit code is in the file sub

At this point, you enter the command CTRL-X CTRL-F. Vim now completes the current word "sub" by looking at the files in the current directory. The first match is sub_count.c. This is not the one you want, so you match the next file by typing CTRL-N. This match is sub_done.c. Typing CTRL-N again takes you to sub_exit.c. The results:

The exit code is in the file sub_exit.c

If the file name starts with / (Unix) or C:\ (MS-Windows) you can find all files in the file system. For example, type "/u" and CTRL-X CTRL-F. This will match "/usr" (this is on Unix):

the file is found in /usr/

If you now press CTRL-N you go back to "/u". Instead, to accept the "/usr/" and go one directory level deeper, use CTRL-X CTRL-F again:

the file is found in /usr/X11R6/

The results depend on what is found in your file system, of course. The matches are sorted alphabetically.

COMPLETING IN SOURCE CODE

Source code files are well structured. That makes it possible to do completion in an intelligent way. In Vim this is called Omni completion. In some other editors it's called intellisense, but that is a trademark.

The key to Omni completion is CTRL-X CTRL-O. Obviously the O stands for Omni here, so that you can remember it easier. Let's use an example for editing C source:

```
{
    struct foo *p;
    p->
```

The cursor is after "p->". Now type CTRL-X CTRL-O. Vim will offer you a list of alternatives, which are the items that "struct foo" contains. That is quite different from using CTRL-P, which would complete any word, while only members of "struct foo" are valid here.

For Omni completion to work you may need to do some setup. At least make sure filetype plugins are enabled. Your vimrc file should contain a line like this:

```
filetype plugin on
```

Or:

```
filetype plugin indent on
```

For C code you need to create a tags file and set the 'tags' option. That is explained ft-c-omni. For other filetypes you may need to do something similar, look below compl-omni-filetypes. It only works for specific filetypes. Check the value of the 'omnifunc' option to find out if it would work.

24.4 Repeating an insert

If you press CTRL-A, the editor inserts the text you typed the last time you were in Insert mode.
   Assume, for example, that you have a file that begins with the following:

```
"file.h" ~
/* Main program begins */ ~
```

You edit this file by inserting "#include " at the beginning of the first line:

```
#include "file.h" ~
/* Main program begins */ ~
```

You go down to the beginning of the next line using the commands "j^". You now start to insert a new "#include" line. So you type:

```
i CTRL-A
```

The result is as follows:

```
#include "file.h" ~
#include /* Main program begins */ ~
```

The "#include " was inserted because CTRL-A inserts the text of the previous insert. Now you type "main.h"<Enter> to finish the line:

```
#include "file.h" ~
#include "main.h" ~
/* Main program begins */ ~
```

The CTRL-@ command does a CTRL-A and then exits Insert mode. That's a quick way of doing exactly the same insertion again.

24.5 Copying from another line

The CTRL-Y command inserts the character above the cursor. This is useful when you are duplicating a previous line. For example, you have this line of C code:

b_array[i]->s_next = a_array[i]->s_next;

Now you need to type the same line, but with "s_prev" instead of "s_next". Start the new line, and press CTRL-Y 14 times, until you are at the "n" of "next":

b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_

Now you type "prev":

b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_prev

Continue pressing CTRL-Y until the following "next":

b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_prev = a_array[i]->s_

Now type "prev;" to finish it off.

The CTRL-E command acts like CTRL-Y except it inserts the character below the cursor.

24.6 Inserting a register

The command CTRL-R {register} inserts the contents of the register. This is useful to avoid having to type a long word. For example, you need to type this:

r = VeryLongFunction(a) + VeryLongFunction(b) + VeryLongFunction(c)

The function name is defined in a different file. Edit that file and move the cursor on top of the function name there, and yank it into register v:

```
"vyiw
```

"v is the register specification, "yiw" is yank-inner-word. Now edit the file where the new line is to be inserted, and type the first letters:

r =

Now use CTRL-R v to insert the function name:

r = VeryLongFunction

You continue to type the characters in between the function name, and use CTRL-R v two times more.

You could have done the same with completion. Using a register is useful when there are many words that start with the same characters.

If the register contains characters such as <BS> or other special characters, they are interpreted as if they had been typed from the keyboard. If you do not want this to happen (you really want the <BS> to be inserted in the text), use the command CTRL-R CTRL-R {register}.

24.7 Abbreviations

An abbreviation is a short word that takes the place of a long one. For example, "ad" stands for "advertisement". Vim enables you to type an abbreviation and then will automatically expand it for you.

To tell Vim to expand "ad" into "advertisement" every time you insert it, use the following command:

```
:iabbrev ad advertisement
```

Now, when you type "ad", the whole word "advertisement" will be inserted into the text. This is triggered by typing a character that can't be part of a word, for example a space:

What Is Entered	What You See
I saw the a	**I saw the a**
I saw the ad	**I saw the ad**
I saw the ad<Space>	**I saw the advertisement<Space>**

The expansion doesn't happen when typing just "ad". That allows you to type a word like "add", which will not get expanded. Only whole words are checked for abbreviations.

ABBREVIATING SEVERAL WORDS

It is possible to define an abbreviation that results in multiple words. For example, to define "JB" as "Jack Benny", use the following command:

```
:iabbrev JB Jack Benny
```

As a programmer, I use two rather unusual abbreviations:

```
:iabbrev #b /****************************************
:iabbrev #e <Space>****************************************/
```

These are used for creating boxed comments. The comment starts with #b, which draws the top line. I then type the comment text and use #e to draw the bottom line.
   Notice that the #e abbreviation begins with a space. In other words, the first two characters are space-star. Usually Vim ignores spaces between the abbreviation and the expansion. To avoid that problem, I spell space as seven characters: <, S, p, a, c, e, >.

> **Note:**
> ":iabbrev" is a long word to type. ":iab" works just as well.
> That's abbreviating the abbreviate command!

FIXING TYPING MISTAKES

It's very common to make the same typing mistake every time. For example, typing "teh" instead of "the". You can fix this with an abbreviation:

```
:abbreviate teh the
```

You can add a whole list of these. Add one each time you discover a common mistake.

LISTING ABBREVIATIONS

The ":abbreviate" command lists the abbreviations:

```
:abbreviate
i  #e             ****************************************/
i  #b            /****************************************
i  JB            Jack Benny
i  ad            advertisement
!  teh           the
```

The "i" in the first column indicates Insert mode. These abbreviations are only active in Insert mode. Other possible characters are:

c	Command-line mode	:cabbrev
!	both Insert and Command-line mode	:abbreviate

Since abbreviations are not often useful in Command-line mode, you will mostly use the ":iabbrev" command. That avoids, for example, that "ad" gets expanded when typing a command like:

```
:edit ad
```

DELETING ABBREVIATIONS

To get rid of an abbreviation, use the ":unabbreviate" command. Suppose you have the following abbreviation:

```
:abbreviate @f fresh
```

You can remove it with this command:

```
:unabbreviate @f
```

While you type this, you will notice that @f is expanded to "fresh". Don't

worry about this, Vim understands it anyway (except when you have an abbreviation for "fresh", but that's very unlikely).
   To remove all the abbreviations:

```
:abclear
```

":unabbreviate" and ":abclear" also come in the variants for Insert mode (":iunabbreviate and ":iabclear") and Command-line mode (":cunabbreviate" and ":cabclear").

REMAPPING ABBREVIATIONS

There is one thing to watch out for when defining an abbreviation: The resulting string should not be mapped. For example:

```
:abbreviate @a adder
:imap dd disk-door
```

When you now type @a, you will get "adisk-doorer". That's not what you want. To avoid this, use the ":noreabbrev" command. It does the same as ":abbreviate", but avoids that the resulting string is used for mappings:

```
:noreabbrev @a adder
```

Fortunately, it's unlikely that the result of an abbreviation is mapped.

24.8 Entering special characters

The CTRL-V command is used to insert the next character literally. In other words, any special meaning the character has, it will be ignored. For example:

```
CTRL-V <Esc>
```

Inserts an escape character. Thus you don't leave Insert mode. (Don't type the space after CTRL-V, it's only to make this easier to read).

> **Note:**
> On MS-Windows CTRL-V is used to paste text. Use CTRL-Q instead of CTRL-V. On Unix, on the other hand, CTRL-Q does not work on some terminals, because it has a special meaning.

You can also use the command CTRL-V {digits} to insert a character with the decimal number {digits}. For example, the character number 127 is the <Del> character (but not necessarily the <Del> key!). To insert <Del> type:

```
CTRL-V 127
```

You can enter characters up to 255 this way. When you type fewer than two digits, a non-digit will terminate the command. To avoid the need of typing a non-digit, prepend one or two zeros to make three digits.
   All the next commands insert a <Tab> and then a dot:

```
CTRL-V 9.
CTRL-V 09.
CTRL-V 009.
```

To enter a character in hexadecimal, use an "x" after the CTRL-V:

```
CTRL-V x7f
```

This also goes up to character 255 (CTRL-V xff). You can use "o" to type a character as an octal number and two more methods allow you to type up to a 16 bit and a 32 bit number (e.g., for a Unicode character):

```
CTRL-V o123
CTRL-V u1234
CTRL-V U12345678
```

24.9 Digraphs

Some characters are not on the keyboard. For example, the copyright character (©). To type these characters in Vim, you use digraphs, where two characters represent one. To enter a ©, for example, you press three keys:

```
CTRL-K Co
```

To find out what digraphs are available, use the following command:

```
:digraphs
```

Vim will display the digraph table. Here are three lines of it:

AC ~_ 159 NS | 160 !I ¡ 161 Ct ¢ 162 Pd £ 163 Cu ¤ 164 Ye ¥ 165
BB ¦ 166 SE § 167 ': ¨ 168 Co © 169 -a ª 170 << « 171 NO ¬ 172
-- 173 Rg ® 174 'm ¯ 175 DG ° 176 +- ± 177 2S ² 178 3S ³ 179

This shows, for example, that the digraph you get by typing CTRL-K Pd is the character (£). This is character number 163 (decimal).
   Pd is short for Pound. Most digraphs are selected to give you a hint about the character they will produce. If you look through the list you will understand the logic.
   You can exchange the first and second character, if there is no digraph for that combination. Thus CTRL-K dP also works. Since there is no digraph for "dP" Vim will also search for a "Pd" digraph.

> **Note:**
> The digraphs depend on the character set that Vim assumes you are using. Always use ":digraphs" to find out which digraphs are currently available.

You can define your own digraphs by specifying the target character with a decimal number. Example:

```
:digraph a\" 228
```

This defines that CTRL-K a" inserts an ä character. **Note:** we had to escape the " character since otherwise it would act as a comment character.

More information about digraphs here: digraphs
   Another way to insert special characters is with a keymap. More about that here: 45.5

24.10 Normal mode commands

Insert mode offers a limited number of commands. In Normal mode you have many more. When you want to use one, you usually leave Insert mode with <Esc>, execute the Normal mode command, and re-enter Insert mode with "i" or "a".
   There is a quicker way. With CTRL-O {command} you can execute any Normal mode command from Insert mode. For example, to delete from the cursor to the end of the line:

```
CTRL-O D
```

You can execute only one Normal mode command this way. But you can specify a register or a count. A more complicated example:

```
CTRL-O "g3dw
```

This deletes up to the third word into register g.

Next chapter: usr_25.txt Editing formatted text

Copyright: see manual-copyright

Editing formatted text

Text hardly ever comes in one sentence per line. This chapter is about
breaking sentences to make them fit on a page and other formatting.
Vim also has useful features for editing single-line paragraphs and tables.

25.1 Breaking lines
25.2 Aligning text
25.3 Indents and tabs
25.4 Dealing with long lines
25.5 Editing tables

Next chapter: usr_26.txt Repeating
Previous chapter: usr_24.txt Inserting quickly
Table of contents: usr_toc.txt

Breaking lines

Vim has a number of functions that make dealing with text easier. By default,
the editor does not perform automatic line breaks. In other words, you have
to press <Enter> yourself. This is useful when you are writing programs where
you want to decide where the line ends. It is not so good when you are
creating documentation and want the text to be at most 70 character wide.
   If you set the 'textwidth' option, Vim automatically inserts line breaks.
Suppose, for example, that you want a very narrow column of only 30
characters. You need to execute the following command:

```
:set textwidth=30
```

Now you start typing (ruler added):

```
         1         2         3
123456789012345678901234567890123456789
I taught programming for a whi
```

If you type "l" next, this makes the line longer than the 30-character limit.
When Vim sees this, it inserts a line break and you get the following:

```
         1         2         3
123456789012345678901234567890123456789
I taught programming for a
whil
```

Continuing on, you can type in the rest of the paragraph:

```
         1         2         3
123456789012345678901234567890123456789
I taught programming for a
while. One time, I was stopped
by the Fort Worth police,
because my homework was too
hard. True story.
```

You do not have to type newlines; Vim puts them in automatically.

        Note:
        The 'wrap' option makes Vim display lines with a line break, but this
        doesn't insert a line break in the file.

REFORMATTING

The Vim editor is not a word processor. In a word processor, if you delete
something at the beginning of the paragraph, the line breaks are reworked. In
Vim they are not; so if you delete the word "programming" from the first line,
all you get is a short line:

```
         1         2         3
123456789012345678901234567890123456789
I taught for a
while. One time, I was stopped
by the Fort Worth police,
because my homework was too
hard. True story.
```

This does not look good. To get the paragraph into shape you use the "gq"
operator.
   Let's first use this with a Visual selection. Starting from the first
line, type:

```
v4jgq
```

"v" to start Visual mode, "4j" to move to the end of the paragraph and then
the "gq" operator. The result is:

```
         1         2         3
12345678901234567890123456789012345
I taught for a while. One
time, I was stopped by the
Fort Worth police, because my
homework was too hard. True
story.
```

Note: there is a way to do automatic formatting for specific types of text layouts, see auto-format.

Since "gq" is an operator, you can use one of the three ways to select the text it works on: With Visual mode, with a movement and with a text object.
   The example above could also be done with "gq4j". That's less typing, but you have to know the line count. A more useful motion command is "}". This moves to the end of a paragraph. Thus "gq}" formats from the cursor to the end of the current paragraph.
   A very useful text object to use with "gq" is the paragraph. Try this:

```
gqap
```

"ap" stands for "a-paragraph". This formats the text of one paragraph (separated by empty lines). Also the part before the cursor.
   If you have your paragraphs separated by empty lines, you can format the whole file by typing this:

```
gggqG
```

"gg" to move to the first line, "gqG" to format until the last line.
   Warning: If your paragraphs are not properly separated, they will be joined together. A common mistake is to have a line with a space or tab. That's a blank line, but not an empty line.

Vim is able to format more than just plain text. See fo-table for how to change this. See the 'joinspaces' option to change the number of spaces used after a full stop.
   It is possible to use an external program for formatting. This is useful if your text can't be properly formatted with Vim's builtin command. See the 'formatprg' option.

25.2 Aligning text

To center a range of lines, use the following command:

```
:{range}center [width]
```

{range} is the usual command-line range. [width] is an optional line width to use for centering. If [width] is not specified, it defaults to the value of 'textwidth'. (If 'textwidth' is 0, the default is 80.)
   For example:

```
:1,5center 40
```

results in the following:

```
  I taught for a while. One
  time, I was stopped by the
Fort Worth police, because my
 homework was too hard. True
           story.
```

RIGHT ALIGNMENT

Similarly, the ":right" command right-justifies the text:

```
:1,5right 37
```

gives this result:

```
    I taught for a while. One
   time, I was stopped by the
Fort Worth police, because my
  homework was too hard. True
                       story.
```

LEFT ALIGNMENT

Finally there is this command:

```
:{range}left [margin]
```

Unlike ":center" and ":right", however, the argument to ":left" is not the
length of the line. Instead it is the left margin. If it is omitted, the
text will be put against the left side of the screen (using a zero margin
would do the same). If it is 5, the text will be indented 5 spaces. For
example, use these commands:

```
:1left 5
:2,5left
```

This results in the following:

```
     I taught for a while. One
time, I was stopped by the
Fort Worth police, because my
homework was too hard. True
story.
```

JUSTIFYING TEXT justify :Justify Justify() package-justify

Vim has no built-in way of justifying text. However, there is a neat macro
package that does the job. To use this package, execute the following
command:

```
:packadd justify
```

Or put this line in your vimrc:

```
:packadd! justify
```

This Vim script file defines a new visual command "_j". To justify a block of
text, highlight the text in Visual mode and then execute "_j".
   Look in the file for more explanations. To go there, do "gf" on this name:
$VIMRUNTIME/pack/dist/opt/justify/plugin/justify.vim.

An alternative is to filter the text through an external program. Example:

```
:%!fmt
```

25.3 Indents and tabs

Indents can be used to make text stand out from the rest. The example texts
in this manual, for example, are indented by eight columns. You would
normally enter this by typing <Tab> at the start of each line. Take this
text:

```
the first line
the second line
```

This is entered by typing <Tab>, some text, <Enter>, <Tab> and more text.
   The 'autoindent' option inserts indents automatically:

```
:set autoindent
```

When a new line is started it gets the same indent as the previous line. In
the above example, pressing the <Tab> key after <Enter> is not needed anymore.

INCREASING INDENT

To increase the amount of indent in a line, use the ">" operator, in Normal
mode. Often this is used as ">>", which adds indent to the current line.
In Insert mode, use <C-t>.
   The amount of indent added is specified with the 'shiftwidth' option. The
default value is 8. To make ">>" insert four columns worth of indent, for
example, type this:

```
:set shiftwidth=4
```

When used on the second line of the example text, this is what you get:

```
the first line
    the second line
```

"4>>" will increase the indent of four lines.

SOFT TAB STOPS

If you want to make indents a multiple of 4, you set 'shiftwidth' to 4. But when pressing a <Tab> you still get 8 columns worth of indent. To change this, set the 'softtabstop' option:

```
:set softtabstop=4
```

Vim now creates invisible tab stops for your cursor every 4 columns; hitting <Tab> jumps to the next stop and inserts the exact mix of spaces or tabs needed.

> **Note:**
> You could set the 'tabstop' option to 4. However, if you edit the file another time, with 'tabstop' set to the default value of 8, it will look wrong. In other programs and when printing the indent will also be wrong. Therefore it is recommended to keep 'tabstop' at eight all the time. That's the standard value everywhere on UNIX-like systems.

25.4 Dealing with long lines

Sometimes you will be editing a file that is wider than the number of columns in the window. When that occurs, Vim wraps the lines so that everything fits on the screen.

If you switch the 'wrap' option off, each line in the file shows up as one line on the screen. Then the ends of the long lines disappear off the screen to the right.

When you move the cursor to a character that can't be seen, Vim will scroll the text to show it. This is like moving a viewport over the text in the horizontal direction.

By default, Vim does not display a horizontal scrollbar in the GUI. If you want to enable one, use the following command:

```
:set guioptions+=b
```

One horizontal scrollbar will appear at the bottom of the Vim window.

If you don't have a scrollbar or don't want to use it, use these commands to scroll the text. The cursor will stay in the same place, but it's moved back into the visible text if necessary.

zh	scroll right
4zh	scroll four characters right
zH	scroll half a window width right
ze	scroll right to put the cursor at the end
zl	scroll left
4zl	scroll four characters left
zL	scroll half a window width left
zs	scroll left to put the cursor at the start

Let's attempt to show this with one line of text. The cursor is on the "w" of "which". The "current window" above the line indicates the text that is currently visible. The "window"s below the text indicate the text that is visible after the command left of it.

```
                      |<-- current window -->|
       some long text, part of which is visible in the window
ze       |<--     window     -->|
zH        |<--     window     -->|
4zh              |<--     window     -->|
zh                  |<--     window     -->|
zl                    |<--     window     -->|
4zl                      |<--     window     -->|
zL                             |<--     window     -->|
zs                            |<--     window     -->|
```

MOVING WITH WRAP OFF

When 'wrap' is off and the text has scrolled horizontally, you can use the following commands to move the cursor to a character you can see. Thus text left and right of the window is ignored. These never cause the text to scroll:

```
g0              to first visible character in this line
g^              to first non-blank visible character in this line
gm              to middle of screen line
gM              to middle of the text in this line
g$              to last visible character in this line

       |<--      window     -->|
some long    text, part of which is visible in one line
         g0  g^    gm      gM g$
```

BREAKING AT WORDS

edit-no-break

When preparing text for use by another program, you might have to make paragraphs without a line break. A disadvantage of using 'nowrap' is that you can't see the whole sentence you are working on. When 'wrap' is on, words are broken halfway, which makes them hard to read.

A good solution for editing this kind of paragraph is setting the 'linebreak' option. Vim then breaks lines at an appropriate place when displaying the line. The text in the file remains unchanged.

Without 'linebreak' text might look like this:

```
+---------------------------------+
|letter generation program for a b|
|ank.  They wanted to send out a s|
|pecial, personalized letter to th|
|eir richest 1000 customers.  Unfo|
|rtunately for the programmer, he |
+---------------------------------+
```

After:

```
:set linebreak
```

it looks like this:

```
+---------------------------------+
|letter generation program for a  |
|bank.  They wanted to send out a |
|special, personalized letter to  |
|their richest 1000 customers.    |
|Unfortunately for the programmer,|
+---------------------------------+
```

Related options:
'breakat' specifies the characters where a break can be inserted.
'showbreak' specifies a string to show at the start of broken line.
Set 'textwidth' to zero to avoid a paragraph to be split.

MOVING BY VISIBLE LINES

The "j" and "k" commands move to the next and previous lines. When used on a long line, this means moving a lot of screen lines at once.

To move only one screen line, use the "gj" and "gk" commands. When a line doesn't wrap they do the same as "j" and "k". When the line does wrap, they move to a character displayed one line below or above.

You might like to use these mappings, which bind these movement commands to the cursor keys:

```
:map <Up> gk
:map <Down> gj
```

TURNING A PARAGRAPH INTO ONE LINE

edit-paragraph-join

If you want to import text into a program like MS-Word, each paragraph should be a single line. If your paragraphs are currently separated with empty lines, this is how you turn each paragraph into a single line:

```
:g/./,/^$/join
```

That looks complicated. Let's break it up in pieces:

```
        :g/./           A ":global" command that finds all lines that contain
                        at least one character.
             ,/^$/      A range, starting from the current line (the non-empty
                        line) until an empty line.
                  join  The ":join" command joins the range of lines together
                        into one line.
```

Starting with this text, containing eight lines broken at column 30:

```
+---------------------------------+
|A letter generation program      |
|for a bank.  They wanted to      |
|send out a special,              |
|personalized letter.             |
|                                 |
|To their richest 1000            |
|customers.  Unfortunately for    |
|the programmer,                  |
+---------------------------------+
```

You end up with two lines:

```
+---------------------------------+
|A letter generation program for a |
|bank.  They wanted to send out a s|
|pecial, personalized letter.      |
|To their richest 1000 customers.  |
|Unfortunately for the programmer, |
+---------------------------------+
```

Note that this doesn't work when the separating line is blank but not empty;
when it contains spaces and/or tabs. This command does work with blank lines:

```
:g/\S/,/^\s*$/join
```

This still requires a blank or empty line at the end of the file for the last
paragraph to be joined.

25.5 Editing tables

Suppose you are editing a table with four columns:

```
nice table        test 1        test 2      test 3
input A           0.534
input B           0.913
```

You need to enter numbers in the third column. You could move to the second
line, use "A", enter a lot of spaces and type the text.
   For this kind of editing there is a special option:

```
set virtualedit=all
```

Now you can move the cursor to positions where there isn't any text. This is
called "virtual space". Editing a table is a lot easier this way.
   Move the cursor by searching for the header of the last column:

```
/test 3
```

Now press "j" and you are right where you can enter the value for "input A".
Typing "0.693" results in:

```
nice table        test 1     test 2      test 3
input A           0.534                  0.693
input B           0.913
```

Vim has automatically filled the gap in front of the new text for you. Now,
to enter the next field in this column use "Bj". "B" moves back to the start
of a white space separated word. Then "j" moves to the place where the next
field can be entered.

> **Note:**
> You can move the cursor anywhere in the display, also beyond the end
> of a line. But Vim will not insert spaces there, until you insert a
> character in that position.

COPYING A COLUMN

You want to add a column, which should be a copy of the third column and
placed before the "test 1" column. Do this in seven steps:

1. Move the cursor to the left upper corner of this column, e.g., with "/test 3".
2. Press CTRL-V to start blockwise Visual mode.
3. Move the cursor down two lines with "2j". You are now in "virtual space" the "input B" line of the "test 3" column.
4. Move the cursor right, to include the whole column in the selection, plus the space that you want between the columns. "9l" should do it.
5. Yank the selected rectangle with "y".
6. Move the cursor to "test 1", where the new column must be placed.
7. Press "P".

The result should be:

```
nice table        test 3    test 1     test 2      test 3
input A           0.693     0.534                  0.693
input B                     0.913
```

Notice that the whole "test 1" column was shifted right, also the line where
the "test 3" column didn't have text.

Go back to non-virtual cursor movements with:

```
:set virtualedit=
```

VIRTUAL REPLACE MODE

The disadvantage of using 'virtualedit' is that it "feels" different. You
can't recognize tabs or spaces beyond the end of line when moving the cursor
around. Another method can be used: Virtual Replace mode.
   Suppose you have a line in a table that contains both tabs and other
characters. Use "rx" on the first tab:

```
inp     0.693   0.534   0.693

       |
   rx  |
       V

inpx0.693   0.534       0.693
```

The layout is messed up. To avoid that, use the "gr" command:

```
inp     0.693   0.534   0.693

       |
  grx  |
       V

inpx    0.693   0.534   0.693
```

What happens is that the "gr" command makes sure the new character takes the
right amount of screen space. Extra spaces or tabs are inserted to fill the
gap. Thus what actually happens is that a tab is replaced by "x" and then
blanks added to make the text after it keep its place. In this case a
tab is inserted.
   When you need to replace more than one character, you use the "R" command
to go to Replace mode (see 04.9). This messes up the layout and replaces
the wrong characters:

```
inp     0       0.534   0.693

        |
 R0.786 |
        V

inp     0.78634 0.693
```

The "gR" command uses Virtual Replace mode. This preserves the layout:

```
inp     0       0.534   0.693

        |
gR0.786 |
        V

inp     0.786   0.534   0.693
```

REFORMATTING TABS IN TABLES

You edit a file that contains tabular data and the original author of the file
decided to align the tabular data using tab characters (instead of spaces).
Alas, they were using tab stops separated by 4 columns and Vim's default
is 8 columns; the table looks wrong! What can be done?
  To fix the appearance without modifying the file, adjust the setting
temporarily:

```
:set tabstop=4
```

This updates the visual layout, but the file itself remains unchanged.
  Another possibility is to permanently reformat the file. For this Vim
provides the :retab command. First, set 'tabstop' to match original layout
(as above), then run:

```
:retab 8
```

The ":retab" command will change 'tabstop' to 8, while changing the text such that it looks the same. It changes spans of white space into tabs and spaces for this. You can now write the file.

Warning: When using ":retab" on a program, it may change white space inside a string constant. Therefore it's a good habit to use "\t" instead of a real tab.

Next chapter: usr_26.txt Repeating

Copyright: see manual-copyright

Repeating

An editing task is hardly ever unstructured. A change often needs to be made several times. In this chapter a number of useful ways to repeat a change will be explained.

26.1 Repeating with Visual mode
26.2 Add and subtract
26.3 Making a change in many files
26.4 Using Vim from a shell script

Next chapter: usr_27.txt Search commands and patterns
Previous chapter: usr_25.txt Editing formatted text
Table of contents: usr_toc.txt

Repeating with Visual mode

Visual mode is very handy for making a change in any sequence of lines. You can see the highlighted text, thus you can check if the correct lines are changed. But making the selection takes some typing. The "gv" command selects the same area again. This allows you to do another operation on the same text.

Suppose you have some lines where you want to change "2001" to "2002" and "2000" to "2001":

The financial results for 2001 are better
than for 2000. The income increased by 50%,
even though 2001 had more rain than 2000.
2000 2001
income 45,403 66,234

First change "2001" to "2002". Select the lines in Visual mode, and use:

```
:s/2001/2002/g
```

Now use "gv" to reselect the same text. It doesn't matter where the cursor is. Then use ":s/2000/2001/g" to make the second change.

Obviously, you can repeat these changes several times.

26.2 Add and subtract

When repeating the change of one number into another, you often have a fixed offset. In the example above, one was added to each year. Instead of typing a substitute command for each year that appears, the CTRL-A command can be used.

Using the same text as above, search for a year:

```
/19[0-9][0-9]\|20[0-9][0-9]
```

Now press CTRL-A. The year will be increased by one:

The financial results for 2002 are better
than for 2000. The income increased by 50%,
even though 2001 had more rain than 2000.
2000 2001
income 45,403 66,234

Use "n" to find the next year, and press "." to repeat the CTRL-A ("." is a bit quicker to type). Repeat "n" and "." for all years that appear.

Adding more than one can be done by prepending the number to CTRL-A. Suppose you have this list:

1. item four ~

2. item five ~

3. item six ~

Move the cursor to "1." and type:

```
3 CTRL-A
```

The "1." will change to "4.". Again, you can use "." to repeat this on the other numbers.

The CTRL-X command does subtraction in a similar way.

The behavior of CTRL-A and CTRL-X depends on the value of 'nrformats'. For example, if you use:

```
:set nrformats+=octal
```

pressing CTRL-A over "007" will increment to "010", because "007" will be identified as an octal number.

26.3 Making a change in many files

Suppose you have a variable called "x_cnt" and you want to change it to
"x_counter". This variable is used in several of your C files. You need to
change it in all files. This is how you do it.
   Put all the relevant files in the argument list:

```
:args *.c
```

This finds all C files and edits the first one. Now you can perform a
substitution command on all these files:

```
:argdo %s/\<x_cnt\>/x_counter/ge | update
```

The ":argdo" command takes an argument that is another command. That command
will be executed on all files in the argument list.
   The "%s" substitute command that follows works on all lines. It finds the
word "x_cnt" with "\<x_cnt\>". The "\<" and "\>" are used to match the whole
word only, and not "px_cnt" or "x_cnt2".
   The flags for the substitute command include "g" to replace all occurrences
of "x_cnt" in the same line. The "e" flag is used to avoid an error message
when "x_cnt" does not appear in the file. Otherwise ":argdo" would abort on
the first file where "x_cnt" was not found.
   The "|" separates two commands. The following "update" command writes the
file only if it was changed. If no "x_cnt" was changed to "x_counter" nothing
happens.

There is also the ":windo" command, which executes its argument in all
windows. And ":bufdo" executes its argument on all buffers. Be careful with
this, because you might have more files in the buffer list than you think.
Check this with the ":buffers" command (or ":ls").

26.4 Using Vim from a shell script

Suppose you have a lot of files in which you need to change the string
"-person-" to "Jones" and then print it. How do you do that? One way is to
do a lot of typing. The other is to write a shell script to do the work.
   The Vim editor does a superb job as a screen-oriented editor when using
Normal mode commands. For batch processing, however, Normal mode commands do
not result in clear, commented command files; so here you will use Ex mode
instead. This mode gives you a nice command-line interface that makes it easy
to put into a batch file. ("Ex command" is just another name for a
command-line (:) command.)
   The Ex mode commands you need are as follows:

```
%s/-person-/Jones/g
write tempfile
quit
```

You put these commands in the file "change.vim". Now to run the editor in
batch mode, use this shell script:

```
for file in *.txt; do
  vim -e -s $file < change.vim
  lpr -r tempfile
done
```

The for-done loop is a shell construct to repeat the two lines in between,
while the $file variable is set to a different file name each time.
   The second line runs the Vim editor in Ex mode (-e argument) on the file
$file and reads commands from the file "change.vim". The -s argument tells
Vim to operate in silent mode. In other words, do not keep outputting the
:prompt, or any other prompt for that matter.
   The "lpr -r tempfile" command prints the resulting "tempfile" and deletes
it (that's what the -r argument does).

READING FROM STDIN

Vim can read text on standard input. Since the normal way is to read commands
there, you must tell Vim to read text instead. This is done by passing the
"-" argument in place of a file. Example:

```
ls | vim -
```

This allows you to edit the output of the "ls" command, without first saving
the text in a file.
   If you use the standard input to read text from, you can use the "-S"
argument to read a script:

```
producer | vim -S change.vim -
```

NORMAL MODE SCRIPTS

If you really want to use Normal mode commands in a script, you can use it like this:

```
vim -s script file.txt ...
```

> **Note:**
> "-s" has a different meaning when it is used without "-e". Here it means to source the "script" as Normal mode commands. When used with "-e" it means to be silent, and doesn't use the next argument as a file name.

The commands in "script" are executed like you typed them. Don't forget that a line break is interpreted as pressing <Enter>. In Normal mode that moves the cursor to the next line.

To create the script you can edit the script file and type the commands. You need to imagine what the result would be, which can be a bit difficult. Another way is to record the commands while you perform them manually. This is how you do that:

```
vim -w script file.txt ...
```

All typed keys will be written to "script". If you make a small mistake you can just continue and remember to edit the script later.

The "-w" argument appends to an existing script. That is good when you want to record the script bit by bit. If you want to start from scratch and start all over, use the "-W" argument. It overwrites any existing file.

Next chapter: usr_27.txt Search commands and patterns

Copyright: see manual-copyright

Search commands and patterns

In chapter 3 a few simple search patterns were mentioned 03.9. Vim can do much more complex searches. This chapter explains the most often used ones. A detailed specification can be found here: pattern
Options affecting how search is done can be found here: search-options

27.1 Ignoring case
27.2 Wrapping around the file end
27.3 Offsets
27.4 Matching multiple times
27.5 Alternatives
27.6 Character ranges
27.7 Character classes
27.8 Matching a line break
27.9 Examples

Next chapter: usr_28.txt Folding
Previous chapter: usr_26.txt Repeating
Table of contents: usr_toc.txt

Ignoring case

By default, Vim's searches are case sensitive. Therefore, "include", "INCLUDE", and "Include" are three different words and a search will match only one of them.
Now switch on the 'ignorecase' option:

```
:set ignorecase
```

Search for "include" again, and now it will match "Include", "INCLUDE" and "InClUDe". (Set the 'hlsearch' option to quickly see where a pattern matches.)
You can switch this off again with:

```
:set noignorecase
```

But let's keep it set, and search for "INCLUDE". It will match exactly the same text as "include" did. Now set the 'smartcase' option:

```
:set ignorecase smartcase
```

If you have a pattern with at least one uppercase character, the search becomes case sensitive. The idea is that you didn't have to type that uppercase character, so you must have done it because you wanted case to match. That's smart!
With these two options set you find the following matches:

pattern	matches
word	word, Word, WORD, WoRd, etc.
Word	Word
WORD	WORD
WoRd	WoRd

CASE IN ONE PATTERN

If you want to ignore case for one specific pattern, you can do this by prepending the "\c" string. Using "\C" will make the pattern to match case. This overrules the 'ignorecase' and 'smartcase' options, when "\c" or "\C" is used their value doesn't matter.

pattern	matches
\Cword	word
\CWord	Word
\cword	word, Word, WORD, WoRd, etc.
\cWord	word, Word, WORD, WoRd, etc.

A big advantage of using "\c" and "\C" is that it sticks with the pattern. Thus if you repeat a pattern from the search history, the same will happen, no matter if 'ignorecase' or 'smartcase' was changed.

> **Note:**
> The use of "\" items in search patterns depends on the 'magic' option. In this chapter we will assume 'magic' is on, because that is the standard and recommended setting. If you would change 'magic', many search patterns would suddenly become invalid.

> **Note:**
> If your search takes much longer than you expected, you can interrupt it with CTRL-C on Unix and CTRL-Break on MS-Windows.

27.2 Wrapping around the file end

By default, a forward search starts searching for the given string at the current cursor location. It then proceeds to the end of the file. If it has not found the string by that time, it starts from the beginning and searches from the start of the file to the cursor location.
   Keep in mind that when repeating the "n" command to search for the next match, you eventually get back to the first match. If you don't notice this you keep searching forever! To give you a hint, Vim displays this message:

search hit BOTTOM, continuing at TOP

If you use the "?" command, to search in the other direction, you get this message:

search hit TOP, continuing at BOTTOM

Still, you don't know when you are back at the first match. One way to see this is by switching on the 'ruler' option:

```
:set ruler
```

Vim will display the cursor position in the lower righthand corner of the window (in the status line if there is one). It looks like this:

101,29   84%

The first number is the line number of the cursor. Remember the line number where you started, so that you can check if you passed this position again.

NOT WRAPPING

To turn off search wrapping, use the following command:

```
:set nowrapscan
```

Now when the search hits the end of the file, an error message displays:

E385: search hit BOTTOM without match for: forever

Thus you can find all matches by going to the start of the file with "gg" and keep searching until you see this message.
   If you search in the other direction, using "?", you get:

E384: search hit TOP without match for: forever

27.3 Offsets

By default, the search command leaves the cursor positioned on the beginning of the pattern. You can tell Vim to leave it some other place by specifying an offset. For the forward search command "/", the offset is specified by appending a slash (/) and the offset:

```
/default/2
```

This command searches for the pattern "default" and then moves to the beginning of the second line past the pattern. Using this command on the paragraph above, Vim finds the word "default" in the first line. Then the cursor is moved two lines down and lands on "an offset".

If the offset is a simple number, the cursor will be placed at the beginning of the line that many lines from the match. The offset number can be positive or negative. If it is positive, the cursor moves down that many lines; if negative, it moves up.

CHARACTER OFFSETS

The "e" offset indicates an offset from the end of the match. It moves the cursor onto the last character of the match. The command:

```
/const/e
```

puts the cursor on the "t" of "const".
   From that position, adding a number moves forward that many characters. This command moves to the character just after the match:

```
/const/e+1
```

A positive number moves the cursor to the right, a negative number moves it to the left. For example:

```
/const/e-1
```

moves the cursor to the "s" of "const".

If the offset begins with "b", the cursor moves to the beginning of the pattern. That's not very useful, since leaving out the "b" does the same thing. It does get useful when a number is added or subtracted. The cursor then goes forward or backward that many characters. For example:

```
/const/b+2
```

Moves the cursor to the beginning of the match and then two characters to the
right. Thus it lands on the "n".

REPEATING

To repeat searching for the previously used search pattern, but with a
different offset, leave out the pattern:

```
/that
//e
```

Is equal to:

```
/that/e
```

To repeat with the same offset:

```
/
```

"n" does the same thing. To repeat while removing a previously used offset:

```
//
```

SEARCHING BACKWARDS

The "?" command uses offsets in the same way, but you must use "?" to separate
the offset from the pattern, instead of "/":

```
?const?e-2
```

The "b" and "e" keep their meaning, they don't change direction with the use
of "?".

START POSITION

When starting a search, it normally starts at the cursor position. When you
specify a line offset, this can cause trouble. For example:

```
/const/-2
```

This finds the next word "const" and then moves two lines up. If you
use "n" to search again, Vim could start at the current position and find the
same "const" match. Then using the offset again, you would be back where you
started. You would be stuck!
   It could be worse: Suppose there is another match with "const" in the next
line. Then repeating the forward search would find this match and move two
lines up. Thus you would actually move the cursor back!

When you specify a character offset, Vim will compensate for this. Thus the
search starts a few characters forward or backward, so that the same match
isn't found again.

27.4 Matching multiple times

The "*" item specifies that the item before it can match any number of times.
Thus:

```
/a*
```

matches "a", "aa", "aaa", etc. But also "" (the empty string), because zero
times is included.
   The "*" only applies to the item directly before it. Thus "ab*" matches
"a", "ab", "abb", "abbb", etc. To match a whole string multiple times, it
must be grouped into one item. This is done by putting "\(" before it and
"\)" after it. Thus this command:

```
/\(ab\)*
```

Matches: "ab", "abab", "ababab", etc. And also "".

To avoid matching the empty string, use "\+". This makes the previous item
match one or more times.

```
/ab\+
```

Matches "ab", "abb", "abbb", etc. It does not match "a" when no "b" follows.

To match an optional item, use "\=". Example:

```
/folders\=
```

Matches "folder" and "folders".

SPECIFIC COUNTS

To match a specific number of items use the form "\{n,m}". "n" and "m" are numbers. The item before it will be matched "n" to "m" times inclusive. Example:

```
/ab\{3,5}
```

matches "abbb", "abbbb" and "abbbbb".
When "n" is omitted, it defaults to zero. When "m" is omitted it defaults to infinity. When ",m" is omitted, it matches exactly "n" times. Examples:

pattern	match count
\{,4}	0, 1, 2, 3 or 4
\{3,}	3, 4, 5, etc.
\{0,1}	0 or 1, same as \=
\{0,}	0 or more, same as *
\{1,}	1 or more, same as \+
\{3}	3

MATCHING AS LITTLE AS POSSIBLE

The items so far match as many characters as they can find. To match as few as possible, use "\{-n,m}". It works the same as "\{n,m}", except that the minimal amount possible is used.
For example, use:

```
/ab\{-1,3}
```

Will match "ab" in "abbb". Actually, it will never match more than one b, because there is no reason to match more. It requires something else to force it to match more than the lower limit.
The same rules apply to removing "n" and "m". It's even possible to remove both of the numbers, resulting in "\{-}". This matches the item before it zero or more times, as few as possible. The item by itself always matches zero times. It is useful when combined with something else. Example:

```
/a.\{-}b
```

This matches "axb" in "axbxb". If this pattern would be used:

```
/a.*b
```

It would try to match as many characters as possible with .*, thus it matches "axbxb" as a whole.

27.5 Alternatives

The "or" operator in a pattern is "\|". Example:

```
/foo\|bar
```

This matches "foo" or "bar". More alternatives can be concatenated:

```
/one\|two\|three
```

Matches "one", "two" and "three".
To match multiple times, the whole thing must be placed in "\(" and "\)":

```
/\(foo\|bar\)\+
```

This matches "foo", "foobar", "foofoo", "barfoobar", etc.
Another example:

```
/end\(if\|while\|for\)
```

This matches "endif", "endwhile" and "endfor".

A related item is "\&". This requires that both alternatives match in the same place. The resulting match uses the last alternative. Example:

```
/forever\&...
```

This matches "for" in "forever". It will not match "fortuin", for example.

27.6 Character ranges

To match "a", "b" or "c" you could use "/a\|b\|c". When you want to match all
letters from "a" to "z" this gets very long. There is a shorter method:

```
/[a-z]
```

The [] construct matches a single character. Inside you specify which
characters to match. You can include a list of characters, like this:

```
/[0123456789abcdef]
```

This will match any of the characters included. For consecutive characters
you can specify the range. "0-3" stands for "0123". "w-z" stands for "wxyz".
Thus the same command as above can be shortened to:

```
/[0-9a-f]
```

To match the "-" character itself make it the first or last one in the range.
These special characters are accepted to make it easier to use them inside a
[] range (they can actually be used anywhere in the search pattern):

```
        \e      <Esc>
        \t      <Tab>
        \r      <CR>
        \b      <BS>
```

There are a few more special cases for [] ranges, see /[] for the whole
story.

COMPLEMENTED RANGE

To avoid matching a specific character, use "^" at the start of the range.
The [] item then matches everything but the characters included. Example:

```
/"[^"]*"
```

```
         "        a double quote
          [^"]    any character that is not a double quote
              *   as many as possible
               "  a double quote again
```

This matches "foo" and "3!x", including the double quotes.

PREDEFINED RANGES

A number of ranges are used very often. Vim provides a shortcut for these.
For example:

```
/\a
```

Finds alphabetic characters. This is equal to using "/[a-zA-Z]". Here are a
few more of these:

item	matches	equivalent	
\d	digit	[0-9]	
\D	non-digit	[^0-9]	
\x	hex digit	[0-9a-fA-F]	
\X	non-hex digit	[^0-9a-fA-F]	
\s	white space	[	] (<Tab> and <Space>)
\S	non-white characters	[^	] (not <Tab> and <Space>)
\l	lowercase alpha	[a-z]	
\L	non-lowercase alpha	[^a-z]	
\u	uppercase alpha	[A-Z]	
\U	non-uppercase alpha	[^A-Z]	

```
        Note:
        Using these predefined ranges works a lot faster than the character
        range it stands for.
        These items can not be used inside [].  Thus "[\d\l]" does NOT work to
        match a digit or lowercase alpha.  Use "\(\d\|\l\)" instead.
```

See /\s for the whole list of these ranges.

27.7 Character classes

The character range matches a fixed set of characters. A character class is
similar, but with an essential difference: The set of characters can be
redefined without changing the search pattern.
   For example, search for this pattern:

```
/\f\+
```

The "\f" item stands for file name characters. Thus this matches a sequence of characters that can be a file name.
   Which characters can be part of a file name depends on the system you are using. On MS-Windows, the backslash is included, on Unix it is not. This is specified with the 'isfname' option. The default value for Unix is:

```
:set isfname
isfname=@,48-57,/,.,-,_,+,,,#,$,%,~,=
```

For other systems the default value is different. Thus you can make a search pattern with "\f" to match a file name, and it will automatically adjust to the system you are using it on.

> **Note:**
> Actually, Unix allows using just about any character in a file name, including white space. Including these characters in 'isfname' would be theoretically correct. But it would make it impossible to find the end of a file name in text. Thus the default value of 'isfname' is a compromise.

The character classes are:

item	matches	option
\i	identifier characters	'isident'
\I	like \i, excluding digits	
\k	keyword characters	'iskeyword'
\K	like \k, excluding digits	
\p	printable characters	
\P	like \p, excluding digits	
\f	file name characters	'isfname'
\F	like \f, excluding digits	

27.8 Matching a line break

Vim can find a pattern that includes a line break. You need to specify where the line break happens, because all items mentioned so far don't match a line break.
   To check for a line break in a specific place, use the "\n" item:

```
/one\ntwo
```

This will match at a line that ends in "one" and the next line starts with "two". To match "one two" as well, you need to match a space or a line break. The item to use for it is _s:

```
/one\_stwo
```

To allow any amount of white space:

```
/one\_s\+two
```

This also matches when "one " is at the end of a line and "   two" at the start of the next one.

"\s" matches white space, "_s" matches white space or a line break. Similarly, "\a" matches an alphabetic character, and "_a" matches an alphabetic character or a line break. The other character classes and ranges can be modified in the same way by inserting a "_".

Many other items can be made to match a line break by prepending "_". For example: "_." matches any character or a line break.

> **Note:**
> _.* matches everything until the end of the file. Be careful with this, it can make a search command very slow.

Another example is _[], a character range that includes a line break:

```
/"\_[^"]*"
```

This finds a text in double quotes that may be split up in several lines.

27.9 Examples

Here are a few search patterns you might find useful. This shows how the items mentioned above can be combined.

FINDING A CALIFORNIA LICENSE PLATE

A sample license plate number is "1MGU103". It has one digit, three uppercase letters and three digits. Directly putting this into a search pattern:

```
/\d\u\u\u\d\d\d
```

Another way is to specify that there are three digits and letters with a
count:

```
/\d\u\{3}\d\{3}
```

Using [] ranges instead:

```
/[0-9][A-Z]\{3}[0-9]\{3}
```

Which one of these you should use? Whichever one you can remember. The
simple way you can remember is much faster than the fancy way that you can't.
If you can remember them all, then avoid the last one, because it's both more
typing and slower to execute.

FINDING AN IDENTIFIER

In C programs (and many other computer languages) an identifier starts with a
letter and further consists of letters and digits. Underscores can be used
too. This can be found with:

```
/\<\h\w*\>
```

"\<" and "\>" are used to find only whole words. "\h" stands for "[A-Za-z_]"
and "\w" for "[0-9A-Za-z_]".

> **Note:**
> "\<" and "\>" depend on the 'iskeyword' option. If it includes "-",
> for example, then "ident-" is not matched. In this situation use:

```
/\w\@<!\h\w*\w\@!
```

> This checks if "\w" does not match before or after the identifier.
> See /\@<! and /\@!.

Next chapter: usr_28.txt Folding

Copyright: see manual-copyright

Folding

Structured text can be separated in sections. And sections in sub-sections.
Folding allows you to display a section as one line, providing an overview.
This chapter explains the different ways this can be done.

28.1 What is folding?
28.2 Manual folding
28.3 Working with folds
28.4 Saving and restoring folds
28.5 Folding by indent
28.6 Folding with markers
28.7 Folding by syntax
28.8 Folding by expression
28.9 Folding unchanged lines
28.10 Which fold method to use?

Next chapter: usr_29.txt Moving through programs
Previous chapter: usr_27.txt Search commands and patterns
Table of contents: usr_toc.txt

What is folding?

Folding is used to show a range of lines in the buffer as a single line on the
screen. Like a piece of paper which is folded to make it shorter:

```
+-------------------------+
| line 1                  |
| line 2                  |
| line 3                  |
|_____________________________ |
\                          \
 \_____________________________\
 / folded lines             /
/_____________________________/
| line 12                 |
| line 13                 |
| line 14                 |
+-------------------------+
```

The text is still in the buffer, unchanged. Only the way lines are displayed
is affected by folding.

The advantage of folding is that you can get a better overview of the
structure of text, by folding lines of a section and replacing it with a line
that indicates that there is a section.

28.2 Manual folding

Try it out: Position the cursor in a paragraph and type:

```
zfap
```

You will see that the paragraph is replaced by a highlighted line. You have
created a fold. zf is an operator and ap a text object selection. You
can use the zf operator with any movement command to create a fold for the
text that it moved over. zf also works in Visual mode.

To view the text again, open the fold by typing:

```
zo
```

And you can close the fold again with:

```
zc
```

All the folding commands start with "z". With some fantasy, this looks like a
folded piece of paper, seen from the side. The letter after the "z" has a
mnemonic meaning to make it easier to remember the commands:

```
zf      F-old creation
zo      O-pen a fold
zc      C-lose a fold
```

Folds can be nested: A region of text that contains folds can be folded
again. For example, you can fold each paragraph in this section, and then
fold all the sections in this chapter. Try it out. You will notice that
opening the fold for the whole chapter will restore the nested folds as they
were, some may be open and some may be closed.

Suppose you have created several folds, and now want to view all the text.
You could go to each fold and type "zo". To do this faster, use this command:

```
zr
```

This will R-educe the folding. The opposite is:

```
zm
```

This folds M-ore. You can repeat "zr" and "zm" to open and close nested folds of several levels.

If you have nested several levels deep, you can open all of them with:

```
zR
```

This R-educes folds until there are none left. And you can close all folds with:

```
zM
```

This folds M-ore and M-ore.

You can quickly disable the folding with the zn command. Then zN brings back the folding as it was. zi toggles between the two. This is a useful way of working:

- create folds to get overview on your file
- move around to where you want to do your work
- do zi to look at the text and edit it
- do zi again to go back to moving around

More about manual folding in the reference manual: fold-manual

28.3 Working with folds

When some folds are closed, movement commands like "j" and "k" move over a fold like it was a single, empty line. This allows you to quickly move around over folded text.

You can yank, delete and put folds as if it was a single line. This is very useful if you want to reorder functions in a program. First make sure that each fold contains a whole function (or a bit less) by selecting the right 'foldmethod'. Then delete the function with "dd", move the cursor and put it with "p". If some lines of the function are above or below the fold, you can use Visual selection:

- put the cursor on the first line to be moved
- hit "V" to start Visual mode
- put the cursor on the last line to be moved
- hit "d" to delete the selected lines.
- move the cursor to the new position and "p"ut the lines there.

It is sometimes difficult to see or remember where a fold is located, thus where a zo command would actually work. To see the defined folds:

```
:set foldcolumn=4
```

This will show a small column on the left of the window to indicate folds. A "+" is shown for a closed fold. A "-" is shown at the start of each open fold and "|" at following lines of the fold.

You can use the mouse to open a fold by clicking on the "+" in the foldcolumn. Clicking on the "-" or a "|" below it will close an open fold.

To open all folds at the cursor line use zO.
To close all folds at the cursor line use zC.
To delete a fold at the cursor line use zd.
To delete all folds at the cursor line use zD.

When in Insert mode, the fold at the cursor line is never closed. That allows you to see what you type!

Folds are opened automatically when jumping around or moving the cursor left or right. For example, the "0" command opens the fold under the cursor (if 'foldopen' contains "hor", which is the default). The 'foldopen' option can be changed to open folds for specific commands. If you want the line under the cursor always to be open, do this:

```
:set foldopen=all
```

Warning: You won't be able to move onto a closed fold then. You might want to
use this only temporarily and then set it back to the default:

```
:set foldopen&
```

You can make folds close automatically when you move out of it:

```
:set foldclose=all
```

This will re-apply 'foldlevel' to all folds that don't contain the cursor.
You have to try it out if you like how this feels. Use zm to fold more and
zr to fold less (reduce folds).

The folding is local to the window. This allows you to open two windows on
the same buffer, one with folds and one without folds. Or one with all folds
closed and one with all folds open.

28.4 Saving and restoring folds

When you abandon a file (starting to edit another one), the state of the folds
is lost. If you come back to the same file later, all manually opened and
closed folds are back to their default. When folds have been created
manually, all folds are gone! To save the folds use the :mkview command:

```
:mkview
```

This will store the settings and other things that influence the view on the
file. You can change what is stored with the 'viewoptions' option.
When you come back to the same file later, you can load the view again:

```
:loadview
```

You can store up to ten views on one file. For example, to save the current
setup as the third view and load the second view:

```
:mkview 3
:loadview 2
```

Note that when you insert or delete lines the views might become invalid.
Also check out the 'viewdir' option, which specifies where the views are
stored. You might want to delete old views now and then.

28.5 Folding by indent

Defining folds with zf is a lot of work. If your text is structured by
giving lower level items a larger indent, you can use the indent folding
method. This will create folds for every sequence of lines with the same
indent. Lines with a larger indent will become nested folds. This works well
with many programming languages.

Try this by setting the 'foldmethod' option:

```
:set foldmethod=indent
```

Then you can use the zm and zr commands to fold more and reduce folding.
It's easy to see on this example text:

```
This line is not indented
        This line is indented once
                This line is indented twice
                This line is indented twice
        This line is indented once
This line is not indented
        This line is indented once
        This line is indented once
```

Note that the relation between the amount of indent and the fold depth depends
on the 'shiftwidth' option. Each 'shiftwidth' worth of indent adds one to the
depth of the fold. This is called a fold level.

When you use the zr and zm commands you actually increase or decrease the
'foldlevel' option. You could also set it directly:

```
:set foldlevel=3
```

This means that all folds with three times a 'shiftwidth' indent or more will
be closed. The lower the foldlevel, the more folds will be closed. When
'foldlevel' is zero, all folds are closed. zM does set 'foldlevel' to zero.
The opposite command zR sets 'foldlevel' to the deepest fold level that is
present in the file.

Thus there are two ways to open and close the folds:
(A) By setting the fold level.
    This gives a very quick way of "zooming out" to view the structure of the
    text, move the cursor, and "zoom in" on the text again.

(B) By using zo and zc commands to open or close specific folds.
    This allows opening only those folds that you want to be open, while other
    folds remain closed.

This can be combined: You can first close most folds by using zm a few times
and then open a specific fold with zo. Or open all folds with zR and
then close specific folds with zc.

But you cannot manually define folds when 'foldmethod' is "indent", as that
would conflict with the relation between the indent and the fold level.

More about folding by indent in the reference manual: fold-indent

28.6 Folding with markers

Markers in the text are used to specify the start and end of a fold region.
This gives precise control over which lines are included in a fold. The
disadvantage is that the text needs to be modified.

Try it:

```
:set foldmethod=marker
```

Example text, as it could appear in a C program:

```
/* foobar () {{{ */
int foobar()
{
        /* return a value {{{ */
        return 42;
        /* }}} */
}
/* }}} */
```

Notice that the folded line will display the text before the marker. This is
very useful to tell what the fold contains.

It's quite annoying when the markers don't pair up correctly after moving some
lines around. This can be avoided by using numbered markers. Example:

```
/* global variables {{{1 */
int varA, varB;

/* functions {{{1 */
/* funcA() {{{2 */
void funcA() {}

/* funcB() {{{2 */
void funcB() {}
/* }}}1 */
```

At every numbered marker a fold at the specified level begins. This will make
any fold at a higher level stop here. You can just use numbered start markers
to define all folds. Only when you want to explicitly stop a fold before
another starts you need to add an end marker.

More about folding with markers in the reference manual: fold-marker

28.7 Folding by syntax

For each language Vim uses a different syntax file. This defines the colors
for various items in the file. If you are reading this in Vim, in a terminal
that supports colors, the colors you see are made with the "help" syntax file.
   In the syntax files it is possible to add syntax items that have the "fold"
argument. These define a fold region. This requires writing a syntax file
and adding these items in it. That's not so easy to do. But once it's done,
all folding happens automatically.
   Here we'll assume you are using an existing syntax file. Then there is
nothing more to explain. You can open and close folds as explained above.
The folds will be created and deleted automatically when you edit the file.

More about folding by syntax in the reference manual: fold-syntax

28.8 Folding by expression

This is similar to folding by indent, but instead of using the indent of a
line a user function is called to compute the fold level of a line. You can
use this for text where something in the text indicates which lines belong
together. An example is an e-mail message where the quoted text is indicated
by a ">" before the line. To fold these quotes use this:

```
:set foldmethod=expr
:set foldexpr=strlen(substitute(substitute(getline(v:lnum),'\\s','',\"g\"),'[^>].*','',''))
```

You can try it out on this text:

> quoted text he wrote
> quoted text he wrote

> > double quoted text I wrote
> > double quoted text I wrote

Explanation for the 'foldexpr' used in the example (inside out):
```
   getline(v:lnum)                      gets the current line
   substitute(...,'\\s','','g')         removes all white space from the line
   substitute(...,'[^>].*','','')       removes everything after leading '>'s
   strlen(...)                          counts the length of the string, which
                                        is the number of '>'s found
```

Note that a backslash must be inserted before every space, double quote and backslash for the ":set" command. If this confuses you, do

```
:set foldexpr
```

to check the actual resulting value. To correct a complicated expression, use the command-line completion:

```
:set foldexpr=<Tab>
```

Where <Tab> is a real Tab. Vim will fill in the previous value, which you can then edit.

When the expression gets more complicated you should put it in a function and set 'foldexpr' to call that function.

More about folding by expression in the reference manual: fold-expr

28.9 Folding unchanged lines

This is useful when you set the 'diff' option in the same window. The -d option does this for you. Example:

```
:setlocal diff foldmethod=diff scrollbind nowrap foldlevel=1
```

Do this in every window that shows a different version of the same file. You will clearly see the differences between the files, while the text that didn't change is folded.

For more details see fold-diff.

28.10 Which fold method to use?

All these possibilities make you wonder which method you should choose. Unfortunately, there is no golden rule. Here are some hints.

If there is a syntax file with folding for the language you are editing, that is probably the best choice. If there isn't one, you might try to write it. This requires a good knowledge of search patterns. It's not easy, but when it's working you will not have to define folds manually.

Typing commands to manually fold regions can be used for unstructured text. Then use the :mkview command to save and restore your folds.

The marker method requires you to change the file. If you are sharing the files with other people or you have to meet company standards, you might not be allowed to add them.
   The main advantage of markers is that you can put them exactly where you want them. That avoids that a few lines are missed when you cut and paste folds. And you can add a comment about what is contained in the fold.

Folding by indent is something that works in many files, but not always very well. Use it when you can't use one of the other methods. However, it is very useful for outlining. Then you specifically use one 'shiftwidth' for each nesting level.

Folding with expressions can make folds in almost any structured text. It is quite simple to specify, especially if the start and end of a fold can easily be recognized.
   If you use the "expr" method to define folds, but they are not exactly how you want them, you could switch to the "manual" method. This will not remove the defined folds. Then you can delete or add folds manually.

Next chapter: usr_29.txt Moving through programs

Copyright: see manual-copyright

Moving through programs

The creator of Vim is a computer programmer. It's no surprise that Vim contains many features to aid in writing programs. Jump around to find where identifiers are defined and used. Preview declarations in a separate window. There is more in the next chapter.

29.1 Using tags
29.2 The preview window
29.3 Moving through a program
29.4 Finding global identifiers
29.5 Finding local identifiers

Next chapter: usr_30.txt Editing programs
Previous chapter: usr_28.txt Folding
Table of contents: usr_toc.txt

Using tags

What is a tag? It is a location where an identifier is defined. An example is a function definition in a C or C++ program. A list of tags is kept in a tags file. This can be used by Vim to directly jump from any place to the tag, the place where an identifier is defined.
   To generate the tags file for all C files in the current directory, use the following command:

```
ctags *.c
```

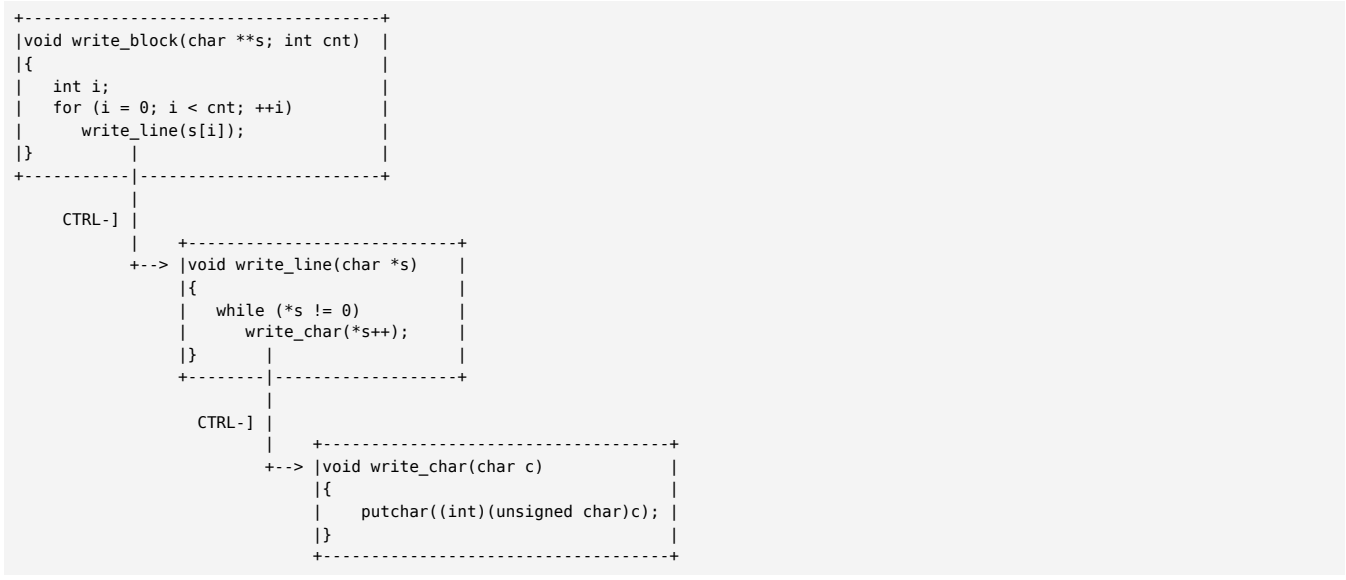
"ctags" is a separate program. Most Unix systems already have it installed. If you do not have it yet, you can find Universal ctags at:
        https://ctags.io

Universal ctags is preferred, Exuberant ctags is no longer being developed.

Now when you are in Vim and you want to go to a function definition, you can jump to it by using the following command:

```
:tag startlist
```

This command will find the function "startlist" even if it is in another file.
   The CTRL-] command jumps to the tag of the word that is under the cursor. This makes it easy to explore a tangle of C code. Suppose, for example, that you are in the function "write_block". You can see that it calls "write_line". But what does "write_line" do? By placing the cursor on the call to "write_line" and pressing CTRL-], you jump to the definition of this function.
   The "write_line" function calls "write_char". You need to figure out what it does. So you position the cursor over the call to "write_char" and press CTRL-]. Now you are at the definition of "write_char".



The ":tags" command shows the list of tags that you traversed through:

```
        :tags
# TO tag           FROM line  in file/text
1  1 write_line    8  write_block.c
2  1 write_char    7  write_line.c
>
```

Now to go back. The CTRL-T command goes to the preceding tag. In the example above you get back to the "write_line" function, in the call to "write_char".
   This command takes a count argument that indicates how many tags to jump back. You have gone forward, and now back. Let's go forward again. The following command goes to the tag on top of the list:

```
:tag
```

You can prefix it with a count and jump forward that many tags. For example:
":3tag". CTRL-T also can be preceded with a count.
   These commands thus allow you to go down a call tree with CTRL-] and back
up again with CTRL-T. Use ":tags" to find out where you are.

SPLIT WINDOWS

The ":tag" command replaces the file in the current window with the one
containing the new function. But suppose you want to see not only the old
function but also the new one? You can split the window using the ":split"
command followed by the ":tag" command. Vim has a shorthand command that does
both:

```
:stag tagname
```

To split the current window and jump to the tag under the cursor use this
command:

```
CTRL-W ]
```

If a count is specified, the new window will be that many lines high.

MORE TAGS FILES

When you have files in many directories, you can create a tags file in each of
them. Vim will then only be able to jump to tags within that directory.
   To find more tags files, set the 'tags' option to include all the relevant
tags files. Example:

```
:set tags=./tags,./../tags,./*/tags
```

This finds a tags file in the same directory as the current file, one
directory level higher and in all subdirectories.
   This is quite a number of tags files, but it may still not be enough. For
example, when editing a file in "~/proj/src", you will not find the tags file
"~/proj/sub/tags". For this situation Vim offers to search a whole directory
tree for tags files. Example:

```
:set tags=~/proj/**/tags
```

ONE TAGS FILE

When Vim has to search many places for tags files, you can hear the disk
rattling. It may get a bit slow. In that case it's better to spend this
time while generating one big tags file. You might do this overnight.
   This requires the Universal or Exuberant ctags program, mentioned above.
It offers an argument to search a whole directory tree:

```
cd ~/proj
ctags -R .
```

The nice thing about this is that Universal/Exuberant ctags recognizes various
file types. Thus this doesn't work just for C and C++ programs, also for
Eiffel and even Vim scripts. See the ctags documentation to tune this.
   Now you only need to tell Vim where your big tags file is:

```
:set tags=~/proj/tags
```

MULTIPLE MATCHES

When a function is defined multiple times (or a method in several classes),
the ":tag" command will jump to the first one. If there is a match in the
current file, that one is used first.
   You can now jump to other matches for the same tag with:

```
:tnext
```

Repeat this to find further matches. If there are many, you can select which
one to jump to:

```
:tselect tagname
```

Vim will present you with a list of choices:

```
  # pri kind tag                  file
  1 F f mch_init os_amiga.c
    mch_init()
  2 F f mch_init os_mac.c
    mch_init()
  3 F f mch_init os_msdos.c
    mch_init(void)
  4 F f mch_init os_riscos.c
    mch_init()
  Enter nr of choice (<CR> to abort):
```

You can now enter the number (in the first column) of the match that you would like to jump to. The information in the other columns give you a good idea of where the match is defined.

To move between the matching tags, these commands can be used:

```
        :tfirst                 go to first match
        :[count]tprevious       go to [count] previous match
        :[count]tnext           go to [count] next match
        :tlast                  go to last match
```

If [count] is omitted then one is used.

GUESSING TAG NAMES

Command line completion is a good way to avoid typing a long tag name. Just type the first bit and press <Tab>:

```
:tag write_<Tab>
```

You will get the first match. If it's not the one you want, press <Tab> until you find the right one.
Sometimes you only know part of the name of a function. Or you have many tags that start with the same string, but end differently. Then you can tell Vim to use a pattern to find the tag.
Suppose you want to jump to a tag that contains "block". First type this:

```
:tag /block
```

Now use command line completion: press <Tab>. Vim will find all tags that contain "block" and use the first match.
The "/" before a tag name tells Vim that what follows is not a literal tag name, but a pattern. You can use all the items for search patterns here. For example, suppose you want to select a tag that starts with "write_":

```
:tselect /^write_
```

The "^" specifies that the tag starts with "write_". Otherwise it would also be found halfway in a tag name. Similarly "$" at the end makes sure the pattern matches until the end of a tag.

A TAGS BROWSER

Since CTRL-] takes you to the definition of the identifier under the cursor, you can use a list of identifier names as a table of contents. Here is an example.
First create a list of identifiers (this requires Universal or Exuberant ctags):

```
ctags --c-types=f -f functions *.c
```

Now start Vim without a file, and edit this file in Vim, in a vertically split window:

```
vim
:vsplit functions
```

The window contains a list of all the functions. There is some more stuff, but you can ignore that. Do ":setlocal ts=99" to clean it up a bit.
In this window, define a mapping:

```
:nnoremap <buffer> <CR> 0ye<C-W>w:tag <C-R>"<CR>
```

Move the cursor to the line that contains the function you want to go to. Now press <Enter>. Vim will go to the other window and jump to the selected function.

RELATED ITEMS

To make case in tag names be ignored, you can set 'ignorecase' while leaving 'tagcase' as "followic", or set 'tagcase' to "ignore".

The 'tagbsearch' option tells if the tags file is sorted or not. The default is to assume a sorted tags file, which makes a tags search a lot faster, but

doesn't work if the tags file isn't sorted.

The 'taglength' option can be used to tell Vim the number of significant characters in a tag.

29.2 The preview window

When you edit code that contains a function call, you need to use the correct arguments. To know what values to pass you can look at how the function is defined. The tags mechanism works very well for this. Preferably the definition is displayed in another window. For this the preview window can be used.

To open a preview window to display the function "write_char":

```
:ptag write_char
```

Vim will open a window, and jumps to the tag "write_char". Then it takes you back to the original position. Thus you can continue typing without the need to use a CTRL-W command.

If the name of a function appears in the text, you can get its definition in the preview window with:

```
CTRL-W }
```

There is a script that automatically displays the text where the word under the cursor was defined. See CursorHold-example.

To close the preview window use this command:

```
:pclose
```

To edit a specific file in the preview window, use ":pedit". This can be useful to edit a header file, for example:

```
:pedit defs.h
```

Finally, ":psearch" can be used to find a word in the current file and any included files and display the match in the preview window. This is especially useful when using library functions, for which you do not have a tags file. Example:

```
:psearch popen
```

This will show the "stdio.h" file in the preview window, with the function prototype for popen():

```
FILE    *popen __P((const char *, const char *));
```

You can specify the height of the preview window, when it is opened, with the 'previewheight' option.

29.3 Moving through a program

Since a program is structured, Vim can recognize items in it. Specific commands can be used to move around.

C programs often contain constructs like this:

```
#ifdef USE_POPEN
    fd = popen("ls", "r")
#else
    fd = fopen("tmp", "w")
#endif
```

But then much longer, and possibly nested. Position the cursor on the "#ifdef" and press %. Vim will jump to the "#else". Pressing % again takes you to the "#endif". Another % takes you to the "#ifdef" again.

When the construct is nested, Vim will find the matching items. This is a good way to check if you didn't forget an "#endif".

When you are somewhere inside a "#if" - "#endif", you can jump to the start of it with:

```
[#
```

If you are not after a "#if" or "#ifdef" Vim will beep. To jump forward to the next "#else" or "#endif" use:

```
]#
```

These two commands skip any "#if" - "#endif" blocks that they encounter.

Example:

```
#if defined(HAS_INC_H)
a = a + inc();
# ifdef USE_THEME
a += 3;
# endif
set_width(a);
```

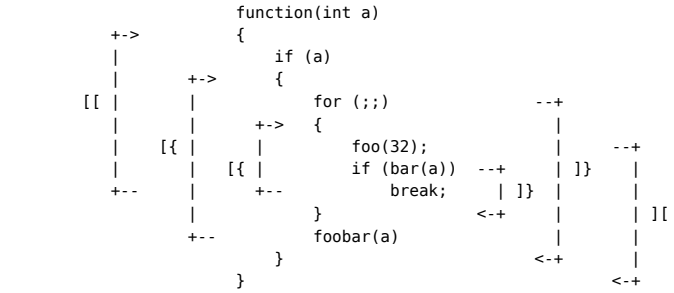
With the cursor in the last line, "[#" moves to the first line. The "#ifdef"
- "#endif" block in the middle is skipped.

MOVING IN CODE BLOCKS

In C code blocks are enclosed in {}. These can get pretty long. To move to the start of the outer block use the "[[" command. Use "][" to find the end. This assumes that the "{" and "}" are in the first column.

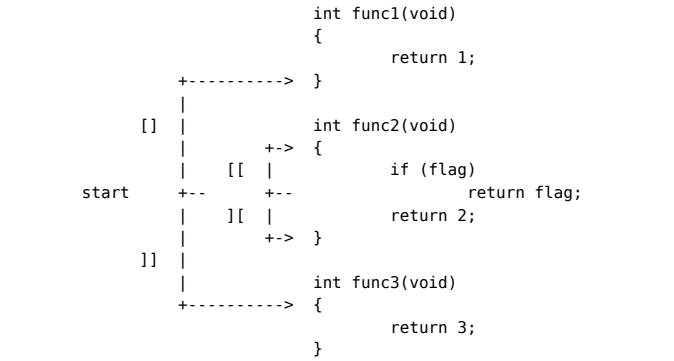
The [{ command moves to the start of the current block. It skips over pairs of {} at the same level. "]}" jumps to the end.

An overview:



When writing C++ or Java, the outer {} block is for the class. The next level of {} is for a method. When somewhere inside a class use "[m" to find the previous start of a method. "]m" finds the next start of a method.

Additionally, "[]" moves backward to the end of a function and "]]" moves forward to the start of the next function. The end of a function is defined by a "}" in the first column.

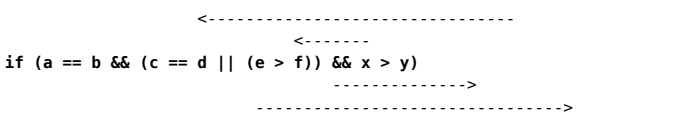


Don't forget you can also use "%" to move between matching (), {} and []. That also works when they are many lines apart.

MOVING IN BRACES

The [(and]) commands work similar to [{ and]}, except that they work on () pairs instead of {} pairs.

```
[(
```




```
])
```

MOVING IN COMMENTS

To move back to the start of a comment use "[/". Move forward to the end of a comment with "]/". This only works for /* - */ comments.

```
   +->    +-> /*
   |    [/ |    * A comment about      --+
[/ |      +--  * wonderful life.        | ]/
   |           */                     <-+
   |
   +--         foo = bar * 3;         --+
                                        | ]/
                /* a short comment */  <-+
```

29.4 Finding global identifiers

You are editing a C program and wonder if a variable is declared as "int" or "unsigned". A quick way to find this is with the "[I" command.
Suppose the cursor is on the word "column". Type:

```
[I
```

Vim will list the matching lines it can find. Not only in the current file, but also in all included files (and files included in them, etc.). The result looks like this:

```
structs.h
 1:   29     unsigned     column;    /* column number */
```

The advantage over using tags or the preview window is that included files are searched. In most cases this results in the right declaration to be found. Also when the tags file is out of date. Also when you don't have tags for the included files.
However, a few things must be right for "[I" to do its work. First of all, the 'include' option must specify how a file is included. The default value works for C and C++. For other languages you will have to change it.

LOCATING INCLUDED FILES

Vim will find included files in the places specified with the 'path' option. If a directory is missing, some include files will not be found. You can discover this with this command:

```
:checkpath
```

It will list the include files that could not be found. Also files included by the files that could be found. An example of the output:

```
--- Included files not found in path ---
<io.h>
vim.h -->
<functions.h>
<clib/exec_protos.h>
```

The "io.h" file is included by the current file and can't be found. "vim.h" can be found, thus ":checkpath" goes into this file and checks what it includes. The "functions.h" and "clib/exec_protos.h" files, included by "vim.h" are not found.

> **Note:**
> Vim is not a compiler. It does not recognize "#ifdef" statements. This means every "#include" statement is used, also when it comes after "#if NEVER".

To fix the files that could not be found, add a directory to the 'path' option. A good place to find out about this is the Makefile. Look out for lines that contain "-I" items, like "-I/usr/local/X11". To add this directory use:

```
:set path+=/usr/local/X11
```

When there are many subdirectories, you can use the "*" wildcard. Example:

```
:set path+=/usr/*/include
```

This would find files in "/usr/local/include" as well as "/usr/X11/include".

When working on a project with a whole nested tree of included files, the "**" items is useful. This will search down in all subdirectories. Example:

```
:set path+=/projects/invent/**/include
```

This will find files in the directories:

```
/projects/invent/include
/projects/invent/main/include
/projects/invent/main/os/include
        etc.
```

There are even more possibilities. Check out the 'path' option for info.
If you want to see which included files are actually found, use this command:

```
:checkpath!
```

You will get a (very long) list of included files, the files they include, and so on. To shorten the list a bit, Vim shows "(Already listed)" for files that were found before and doesn't list the included files in there again.

JUMPING TO A MATCH

"[I" produces a list with only one line of text. When you want to have a closer look at the first item, you can jump to that line with the command:

```
[<Tab>
```

You can also use "[CTRL-I", since CTRL-I is the same as pressing <Tab>.

The list that "[I" produces has a number at the start of each line. When you want to jump to another item than the first one, type the number first:

```
3[<Tab>
```

Will jump to the third item in the list. Remember that you can use CTRL-O to jump back to where you started from.

RELATED COMMANDS

```
        [i              only lists the first match
        ]I              only lists items below the cursor
        ]i              only lists the first item below the cursor
```

FINDING DEFINED IDENTIFIERS

The "[I" command finds any identifier. To find only macros, defined with "#define" use:

```
[D
```

Again, this searches in included files. The 'define' option specifies what a line looks like that defines the items for "[D". You could change it to make it work with other languages than C or C++.

The commands related to "[D" are:

```
        [d              only lists the first match
        ]D              only lists items below the cursor
        ]d              only lists the first item below the cursor
```

29.5 Finding local identifiers

The "[I" command searches included files. To search in the current file only, and jump to the first place where the word under the cursor is used:

```
gD
```

Hint: Goto Definition. This command is very useful to find a variable or function that was declared locally ("static", in C terms). Example (cursor on "counter"):

```
    +->   static int counter = 0;
    |
    |     int get_counter(void)
 gD |     {
    |         ++counter;
    +--       return counter;
          }
```

To restrict the search even further, and look only in the current function, use this command:

```
gd
```

This will go back to the start of the current function and find the first occurrence of the word under the cursor. Actually, it searches backwards to an empty line above a "{" in the first column. From there it searches forward for the identifier. Example (cursor on "idx"):

```
         int find_entry(char *name)
         {
    +->      int idx;
    |
 gd |        for (idx = 0; idx < table_len; ++idx)
    |            if (strcmp(table[idx].name, name) == 0)
    +--             return idx;
         }
```

Next chapter: usr_30.txt Editing programs

Copyright: see manual-copyright

Editing programs

Vim has various commands that aid in writing computer programs. Compile a program and directly jump to reported errors. Automatically set the indent for many languages and format comments.

30.1 Compiling
30.2 Indenting C files
30.3 Automatic indenting
30.4 Other indenting
30.5 Tabs and spaces
30.6 Formatting comments

Next chapter: usr_31.txt Exploiting the GUI
Previous chapter: usr_29.txt Moving through programs
Table of contents: usr_toc.txt

Compiling

Vim has a set of so called "quickfix" commands. They enable you to compile a program from within Vim and then go through the errors generated and fix them (hopefully). You can then recompile and fix any new errors that are found until finally your program compiles without any error.

The following command runs the program "make" (supplying it with any argument you give) and captures the results:

```
:make {arguments}
```

If errors were generated, they are captured and the editor positions you where the first error occurred.
Take a look at an example ":make" session. (Typical :make sessions generate far more errors and fewer stupid ones.) After typing ":make" the screen looks like this:

```
:!make | &tee /tmp/vim215953.err
gcc -g -Wall -o prog main.c sub.c
main.c: In function 'main':
main.c:6: too many arguments to function 'do_sub'
main.c: At top level:
main.c:10: parse error before '}'
make: *** [prog] Error 1

2 returned
"main.c" 11L, 111C
(3 of 6): too many arguments to function 'do_sub'
Press ENTER or type command to continue
```

From this you can see that you have errors in the file "main.c". When you press <Enter>, Vim displays the file "main.c", with the cursor positioned on line 6, the first line with an error. You did not need to specify the file or the line number, Vim knew where to go by looking in the error messages.

```
          +-------------------------------------------------+
          |int main()                                       |
          |{                                                |
          |     int i=3;                                    |
cursor -> |     do_sub("foo");                              |
          |     ++i;                                        |
          |     return (0);                                 |
          |}                                                |
          |}                                                |
          | ~                                               |
          |(3 of 12): too many arguments to function 'do_sub' |
          +-------------------------------------------------+
```

The following command goes to where the next error occurs:

```
:cnext
```

Vim jumps to line 10, the last line in the file, where there is an extra '}'.
When there is not enough room, Vim will shorten the error message. To see the whole message use:

```
:cc
```

You can get an overview of all the error messages with the ":clist" command. The output looks like this:

```
:clist
```

```
3 main.c: 6:too many arguments to function 'do_sub'
5 main.c: 10:parse error before '}'
```

Only the lines where Vim recognized a file name and line number are listed here. It assumes those are the interesting lines and the rest is just boring messages. However, sometimes unrecognized lines do contain something you want to see. Output from the linker, for example, about an undefined function. To see all the messages add a "!" to the command:

```
:clist!
```

```
1 gcc -g -Wall -o prog main.c sub.c
2 main.c: In function 'main':
3 main.c:6: too many arguments to function 'do_sub'
4 main.c: At top level:
5 main.c:10: parse error before '}'
6 make: *** [prog] Error 1
```

Vim will highlight the current error. To go back to the previous error, use:

```
:cprevious
```

Other commands to move around in the error list:

```
        :cfirst         to first error
        :clast          to last error
        :cc 3           to error nr 3
```

USING ANOTHER COMPILER

The name of the program to run when the ":make" command is executed is defined by the 'makeprg' option. Usually this is set to "make", but Visual C++ users should set this to "nmake" by executing the following command:

```
:set makeprg=nmake
```

You can also include arguments in this option. Special characters need to be escaped with a backslash. Example:

```
:set makeprg=nmake\ -f\ project.mak
```

You can include special Vim keywords in the command specification. The % character expands to the name of the current file. So if you execute the command:

```
:set makeprg=make\ %:S
```

When you are editing main.c, then ":make" executes the following command:

```
make main.c
```

This is not too useful, so you will refine the command a little and use the :r (root) modifier:

```
:set makeprg=make\ %:r:S.o
```

Now the command executed is as follows:

```
make main.o
```

More about these modifiers here: filename-modifiers.

OLD ERROR LISTS

Suppose you ":make" a program. There is a warning message in one file and an error message in another. You fix the error and use ":make" again to check if it was really fixed. Now you want to look at the warning message. It doesn't show up in the last error list, since the file with the warning wasn't compiled again. You can go back to the previous error list with:

```
:colder
```

Then use ":clist" and ":cc {nr}" to jump to the place with the warning.
   To go forward to the next error list:

```
:cnewer
```

Vim remembers ten error lists.

SWITCHING COMPILERS

You have to tell Vim what format the error messages are that your compiler produces. This is done with the 'errorformat' option. The syntax of this option is quite complicated and it can be made to fit almost any compiler. You can find the explanation here: errorformat.

You might be using various different compilers. Setting the 'makeprg' option, and especially the 'errorformat' each time is not easy. Vim offers a simple method for this. For example, to switch to using the Microsoft Visual C++ compiler:

```
:compiler msvc
```

This will find the Vim script for the "msvc" compiler and set the appropriate options.
   You can write your own compiler files. See write-compiler-plugin.

OUTPUT REDIRECTION

The ":make" command redirects the output of the executed program to an error file. How this works depends on various things, such as the 'shell'. If your ":make" command doesn't capture the output, check the 'makeef' and 'shellpipe' options. The 'shellquote' and 'shellxquote' options might also matter.

In case you can't get ":make" to redirect the file for you, an alternative is to compile the program in another window and redirect the output into a file. Then have Vim read this file with:

```
:cfile {filename}
```

Jumping to errors will work like with the ":make" command.

30.2 Indenting C style text

A program is much easier to understand when the lines have been properly indented. Vim offers various ways to make this less work. For C or C style programs like Java or C++, set the 'cindent' option. Vim knows a lot about C programs and will try very hard to automatically set the indent for you. Set the 'shiftwidth' option to the amount of spaces you want for a deeper level. Four spaces will work fine. One ":set" command will do it:

```
:set cindent shiftwidth=4
```

With this option enabled, when you type something such as "if (x)", the next line will automatically be indented an additional level.

```
                                if (flag)
        Automatic indent   --->     do_the_work();
        Automatic unindent <--      if (other_flag) {
        Automatic indent   --->         do_file();
        keep indent                     do_some_more();
        Automatic unindent <--      }
```

When you type something in curly braces ({}), the text will be indented at the start and unindented at the end. The unindenting will happen after typing the '}', since Vim can't guess what you are going to type.

One side effect of automatic indentation is that it helps you catch errors in your code early. When you type a } to finish a function, only to find that the automatic indentation gives it more indent than what you expected, there is probably a } missing. Use the "%" command to find out which { matches the } you typed.
   A missing) and ; also cause extra indent. Thus if you get more white space than you would expect, check the preceding lines.

When you have code that is badly formatted, or you inserted and deleted lines, you need to re-indent the lines. The "=" operator does this. The simplest form is:

```
==
```

This indents the current line. Like with all operators, there are three ways to use it. In Visual mode "=" indents the selected lines. A useful text object is "a{". This selects the current {} block. Thus, to re-indent the code block the cursor is in:

```
=a{
```

If you have really badly indented code, you can re-indent the whole file with:

```
gg=G
```

However, don't do this in files that have been carefully indented manually. The automatic indenting does a good job, but in some situations you might want to overrule it.

SETTING INDENT STYLE

Different people have different styles of indentation. By default Vim does a pretty good job of indenting in a way that 90% of programmers do. There are different styles, however; so if you want to, you can customize the indentation style with the 'cinoptions' option.
   By default 'cinoptions' is empty and Vim uses the default style. You can add various items where you want something different. For example, to make curly braces be placed like this:

if (flag)

{
i = 8;
j = 0;
}

Use this command:

```
:set cinoptions+={2
```

There are many of these items. See cinoptions-values.

30.3 Automatic indenting

You don't want to switch on the 'cindent' option manually every time you edit a C file. This is how you make it work automatically:

```
:filetype indent on
```

Actually, this does a lot more than switching on 'cindent' for C files. First of all, it enables detecting the type of a file. That's the same as what is used for syntax highlighting.
   When the filetype is known, Vim will search for an indent file for this type of file. The Vim distribution includes a number of these for various programming languages. This indent file will then prepare for automatic indenting specifically for this file.

If you don't like the automatic indenting, you can switch it off again:

```
:filetype indent off
```

If you don't like the indenting for one specific type of file, this is how you avoid it. Create a file with just this one line:

```
:let b:did_indent = 1
```

Now you need to write this in a file with a specific name:

        {directory}/indent/{filetype}.vim

The {filetype} is the name of the file type, such as "cpp" or "java". You can see the exact name that Vim detected with this command:

```
:set filetype
```

In this file the output is:

filetype=help

Thus you would use "help" for {filetype}.
   For the {directory} part you need to use your runtime directory. Look at the output of this command:

```
set runtimepath
```

Now use the first item, the name before the first comma. Thus if the output looks like this:

runtimepath=~/.config/nvim,/usr/local/share/vim/vim60/runtime,~/.config/nvim/after

You use "~/.config/nvim" for {directory}. Then the resulting file name is:

~/.config/nvim/indent/help.vim

Instead of switching the indenting off, you could write your own indent file. How to do that is explained here: indent-expression.

30.4 Other indenting

The simplest form of automatic indenting is with the 'autoindent' option. It uses the indent from the previous line. A bit smarter is the 'smartindent' option. This is useful for languages where no indent file is available. 'smartindent' is not as smart as 'cindent', but smarter than 'autoindent'.
   With 'smartindent' set, an extra level of indentation is added for each { and removed for each }. An extra level of indentation will also be added for any of the words in the 'cinwords' option. Lines that begin with # are treated specially: all indentation is removed. This is done so that preprocessor directives will all start in column 1. The indentation is restored for the next line.

CORRECTING INDENTS

When you are using 'autoindent' or 'smartindent' to get the indent of the previous line, there will be many times when you need to add or remove one 'shiftwidth' worth of indent. A quick way to do this is using the CTRL-D and CTRL-T commands in Insert mode.
   For example, you are typing a shell script that is supposed to look like this:

if test -n a; then
echo a

echo "-------"
fi

Start off by setting these options:

```
:set autoindent shiftwidth=3
```

You start by typing the first line, <Enter> and the start of the second line:

if test -n a; then
echo

Now you see that you need an extra indent. Type CTRL-T. The result:

if test -n a; then
echo

The CTRL-T command, in Insert mode, adds one 'shiftwidth' to the indent, no matter where in the line you are.
   You continue typing the second line, <Enter> and the third line. This time the indent is OK. Then <Enter> and the last line. Now you have this:

if test -n a; then
echo a
echo "-------"
fi

To remove the superfluous indent in the last line press CTRL-D. This deletes one 'shiftwidth' worth of indent, no matter where you are in the line.
   When you are in Normal mode, you can use the ">>" and "<<" commands to shift lines. ">" and "<" are operators, thus you have the usual three ways to specify the lines you want to indent. A useful combination is:

```
>i{
```

This adds one indent to the current block of lines, inside {}. The { and } lines themselves are left unmodified. ">a{" includes them. In this example the cursor is on "printf":

original text after ">i{" after ">a{"

if (flag)	if (flag)	if (flag)
{ { { **printf("yes"); printf("yes"); printf("yes");** **flag = 0; flag = 0; flag = 0;** **} } }**		

30.5 Tabs and spaces

A QUICK HISTORY OF THE RATIONALE BEHIND TABS

vi (the ancestor of Vim) was created by Bill Joy. At the time, he was using a PDP-11 with limited memory and I/O operation capabilities. Back then, it was common to optimize the size of source code with the following trick.
  The ASCII table was first designed to remotely control teleprinters. When control character 9 (the Horizontal Tab, caret notation: ^I) was sent to a teleprinter, it would move the carriage to the next tab stop. Assuming tab stops were separated by 8 columns (a typical standard), this means that a single control character could produce the same visual effect as up to 8 space characters. For example, the following two lines will display identically

```
1234^I9
1234    9
```

Using the <Tab> key was also faster than typing <Space> several times; the same was true for <BS>.

THE ISSUE WITH TABS AND INDENTATION

In Vim, the number of columns between two (virtual) horizontal tab stops is controlled by 'tabstop' and is set to eight by default. Although you can change it, you quickly run into trouble later. Other programs won't know what tabstop value you used. They probably use the default value of eight, and your text suddenly looks very different. Also, most printers use a fixed tabstop value of eight. Thus it's best to keep 'tabstop' alone; if you edit a file which was written with a different tabstop setting, see 25.3 for how to fix that.
   For indenting lines in a program, using a multiple of eight columns makes you quickly run into the right border of the window. Using a single space doesn't provide enough visual difference. Many people prefer to use four spaces, a good compromise.
   Since a tab character at the beginning of a line is visually represented as eight spaces and you want to use an indent of four spaces, you can't use a tab character to make your indent.
  To remedy this, vi had the 'shiftwidth' option. When set to 4, on a new line, pressing <C-t> in Insert mode would indent the line by 4 spaces, a result impossible to get with the <Tab> key and 'tabstop' set to 8.
 To optimize space, vi would also silently remove packs of spaces and replace them with tab characters. The following shows what happens pressing <C-t> a few times.
  A "." stands for a space character and "------->" for a tab character.

type	result
<C-t>	....
<C-t><C-t>	------->
<C-t><C-t><C-t>	------->....

Similarly pressing <C-d> in Insert mode would decrease the indent. Hence with set tabstop=8 shiftwidth=2 one has

type	result
<C-t><Tab><C-t>	..----->..
<C-t><Tab><C-t><C-d>	------->

A third option that one could set in vi was 'autoindent'. It copies the indent level of the previous lines,

type	result
<Space><Tab>hello	.------>hello
<Space><Tab>hello<Enter>	.------>hello ------->

but the new line is produced by optimizing the number of characters used.

JUST SPACES

But separating tab stops with 8 columns was not universal: IBM had a standard at 10 columns, and today some Go developers write code with tabstop=4. Every time text is displayed with a different 'tabstop' value, it risks misaligning the text, especially once the file is shared and opened on another machine.

In the meantime, computers got much better and the few octets saved by using tabs were no longer making any real difference. It became possible to use only spaces and thus guarantee the same resulting text everywhere. But using only spaces was impossible in vi without sacrificing features. Remember that 'autoindent' would systematically try to input a tab character when it could.

Vim 4.0 made working with only spaces as convenient as working only with tabs (or a mix of tabs and spaces), by introducing the 'expandtab' option. When set, Vim will replace any horizontal tab character it would normally insert with an equivalent number of spaces, to end up with the same visual effect. <BS> would continue to remove only one character at a time.

type	result
<Tab>	....
<Tab><BS>	....

CHANGING TABS IN SPACES (AND BACK)

Setting 'expandtab' does not immediately affect existing tab characters. In order to purge a file from all its horizontal tab characters, Vim 5.3 introduced the :retab command. Use these commands:

```
:set expandtab
:retab
```

This is a little bit dangerous, because it can also change tabs inside a string. To check if these exist, you could use this:

```
/"[^"\t]*\t[^"]*"
```

It's recommended not to use actual tab characters inside a string. Replace them with "\t" to avoid trouble.

The other way around works just as well:

```
:set noexpandtab
:retab!
```

SOFT TAB STOPS

When using only spaces, or a mix of spaces and horizontal tabs, one gets the unpleasant feeling that the two keys <Tab> and <BS> do not act in mirror, as they do when using only tab characters.

Vim 5.4 introduced the 'softtabstop' option. On top of the (hard) tab stops used to display the horizontal tab characters in the text, Vim adds extra soft tab stops dedicated only to the cursor. When 'softtabstop' is set to a positive value, the <Tab> key will push the cursor to the next soft tab stop. Vim will insert the correct combination of tab characters and spaces to achieve this effect. Likewise pressing <BS> will have the cursor try to reach the nearest soft tab stop. The following example uses
:set softtabstop=4

type	result
<Tab>	....
<Tab><Tab>a	------->a
<Tab><Tab>a<Tab>	------->a...
<Tab><Tab>a<Tab><BS>	------->a

To maintain global coherence, one can :set softtabstop=-1 so that the value of 'shiftwidth' is used for the number of columns between two soft tab stops.

If you prefer to have different values for 'shiftwidth' and 'softtabstop', you can still do so and use <C-t> to indent with 'shiftwidth'. Or you can use the 'smarttab' option, allowing for a unified <Tab> key that knows what to do in the different situations.

VARIABLE TAB STOPS

As we said before, the ASCII table was designed to remotely control teleprinters. A given teleprinter could be configured to have their physical tab stops have variable spacing. After all, the ^I control character was only stipulating: go to the next tab stop wherever it is.
Vim 7.3 introduced 'vartabstop' to emulate the same functionality. For example if Vim was compiled with +vartabs and :set vartabstop=2,4 one gets

actual character	result
^I	->
^I^I	->--->
^I^I^I	->--->--->

Similarly, 'varsofttabstop' was also introduced, to have variably spaced soft tab stops. With :set varsofttabstop=2,4 one gets

type	result
<Tab>	..
<Tab><Tab>	.....
<Tab><Tab><Tab>	------>....

EXAMPLES OF CONFIGURATION

By default, Vim is configured to use only tabs:

```
:set tabstop=8
:set shiftwidth=8
:set noexpandtab
:set softtabstop=0
:set nosmarttab
```

If you want to write C code as if it were Python (only spaces, with indents of 4 spaces), here is what you can use:

```
:set shiftwidth=4
:set softtabstop=-1
:set expandtab
```

If you want the same behavior but with better control over alignment (e.g. lining up parameters or comments in multiples of 2 spaces), use:

```
:set shiftwidth=4
:set softtabstop=2
:set expandtab
:set smarttab
```

If instead, you would like to write C code like Bram Moolenaar would have (using a mix of tabs and spaces), you can use

```
:set shiftwidth=4
:set softtabstop=-1
```

30.6 Formatting comments

One of the great things about Vim is that it understands comments. You can ask Vim to format a comment and it will do the right thing.
Suppose, for example, that you have the following comment:

```
/*
 * This is a test
 * of the text formatting.
 */
```

You then ask Vim to format it by positioning the cursor at the start of the comment and type:

```
gq]/
```

"gq" is the operator to format text. "]/" is the motion that takes you to the
end of a comment. The result is:

```
/*
 * This is a test of the text formatting.
 */
```

Notice that Vim properly handled the beginning of each line.
  An alternative is to select the text that is to be formatted in Visual mode
and type "gq".

To add a new line to the comment, position the cursor on the middle line and
press "o". The result looks like this:

```
/*
 * This is a test of the text formatting.
 *
 */
```

Vim has automatically inserted a star and a space for you. Now you can type
the comment text. When it gets longer than 'textwidth', Vim will break the
line. Again, the star is inserted automatically:

```
/*
 * This is a test of the text formatting.
 * Typing a lot of text here will make Vim
 * break
 */
```

For this to work some flags must be present in 'formatoptions':

```
        r       insert the star when typing <Enter> in Insert mode
        o       insert the star when using "o" or "O" in Normal mode
        c       break comment text according to 'textwidth'
```

See fo-table for more flags.

DEFINING A COMMENT

The 'comments' option defines what a comment looks like. Vim distinguishes
between a single-line comment and a comment that has a different start, end
and middle part.
   Many single-line comments start with a specific character. In C++ // is
used, in Makefiles #, in Vim scripts ". For example, to make Vim understand
C++ comments:

```
:set comments=://
```

The colon separates the flags of an item from the text by which the comment is
recognized. The general form of an item in 'comments' is:

```
        {flags}:{text}
```

The {flags} part can be empty, as in this case.
   Several of these items can be concatenated, separated by commas. This
allows recognizing different types of comments at the same time. For example,
let's edit an e-mail message. When replying, the text that others wrote is
preceded with ">" and "!" characters. This command would work:

```
:set comments=n:>,n:!
```

There are two items, one for comments starting with ">" and one for comments
that start with "!". Both use the flag "n". This means that these comments
nest. Thus a line starting with ">" may have another comment after the ">".
This allows formatting a message like this:

```
> ! Did you see that site?
> ! It looks really great.
> I don't like it.  The
> colors are terrible.
What is the URL of that
site?
```

Try setting 'textwidth' to a different value, e.g., 80, and format the text by
Visually selecting it and typing "gq". The result is:

```
> ! Did you see that site?  It looks really great.
> I don't like it.  The colors are terrible.
What is the URL of that site?
```

You will notice that Vim did not move text from one type of comment to
another. The "I" in the second line would have fit at the end of the first
line, but since that line starts with "> !" and the second line with ">", Vim
knows that this is a different kind of comment.

A THREE PART COMMENT

A C comment starts with "/*", has "*" in the middle and "*/" at the end. The
entry in 'comments' for this looks like this:

```
:set comments=s1:/*,mb:*,ex:*/
```

The start is defined with "s1:/*". The "s" indicates the start of a
three-piece comment. The colon separates the flags from the text by which the
comment is recognized: "/*". There is one flag: "1". This tells Vim that the
middle part has an offset of one space.
   The middle part "mb:*" starts with "m", which indicates it is a middle
part. The "b" flag means that a blank must follow the text. Otherwise Vim
would consider text like "*pointer" also to be the middle of a comment.
   The end part "ex:*/" has the "e" for identification. The "x" flag has a
special meaning. It means that after Vim automatically inserted a star,
typing / will remove the extra space.

For more details see format-comments.

Next chapter: usr_31.txt Exploiting the GUI

Copyright: see manual-copyright

VIM USER MANUAL by Bram Moolenaar

Exploiting the GUI

Vim works well in a terminal, but the GUI has a few extra items. A file
browser can be used for commands that use a file. A dialog to make a choice
between alternatives. Use keyboard shortcuts to access menu items quickly.

31.1 The file browser
31.2 Confirmation
31.3 Menu shortcuts
31.4 Vim window position and size
31.5 Various

Next chapter: usr_32.txt The undo tree
Previous chapter: usr_30.txt Editing programs
Table of contents: usr_toc.txt

The file browser

You can open the current directory with:

```
:browse edit
```

The same works when opening a split or tabpage:

```
:browse split
:browse vsplit
:browse tabedit
:browse tabnew
```

By default this is handled by the builtin dir plugin. To use a third-party
directory browser instead, see dir-disable. Use the dir browser's mappings
to open a file.

When a file or directory argument is given, the command is executed normally.
For example:

```
:browse edit /etc
```

31.2 Confirmation

Vim protects you from accidentally overwriting a file and other ways to lose
changes. If you do something that might be a bad thing to do, Vim produces an
error message and suggests appending ! if you really want to do it.
   To avoid retyping the command with the !, you can make Vim give you a
dialog. You can then press "OK" or "Cancel" to tell Vim what you want.
   For example, you are editing a file and made changes to it. You start
editing another file with:

```
:confirm edit foo.txt
```

Vim will pop up a dialog that looks something like this:

```
        +----------------------------------+
        |                                  |
        |   ?   Save changes to "bar.txt"?  |
        |                                  |
        |   YES   NO             CANCEL     |
        +----------------------------------+
```

Now make your choice. If you do want to save the changes, select "YES". If
you want to lose the changes for ever: "NO". If you forgot what you were
doing and want to check what really changed use "CANCEL". You will be back in
the same file, with the changes still there.

Just like ":browse", the ":confirm" command can be prepended to most commands
that edit another file. They can also be combined:

```
:confirm browse edit
```

This will produce a dialog when the current buffer was changed. Then it will
pop up a file browser to select the file to edit.

        Note:
        In the dialog you can use the keyboard to select the choice.
        Typically the <Tab> key and the cursor keys change the choice.
        Pressing <Enter> selects the choice. This depends on the system
        though.

When you are not using the GUI, the ":confirm" command works as well. Instead
of popping up a dialog, Vim will print the message at the bottom of the Vim
window and ask you to press a key to make a choice.

```
:confirm edit main.c
```

Save changes to "Untitled"?
[Y]es, (N)o, (C)ancel:

You can now press the single key for the choice. You don't have to press
<Enter>, unlike other typing on the command line.

31.3 Menu shortcuts

The keyboard is used for all Vim commands. The menus provide a simple way to
select commands, without knowing what they are called. But you have to move
your hand from the keyboard and grab the mouse.
   Menus can often be selected with keys as well. This depends on your
system, but most often it works this way. Use the <Alt> key in combination
with the underlined letter of a menu. For example, <A-w> (<Alt> and w) pops
up the Window menu.
   In the Window menu, the "split" item has the p underlined. To select it,
let go of the <Alt> key and press p.

After the first selection of a menu with the <Alt> key, you can use the cursor
keys to move through the menus. <Right> selects a submenu and <left> closes
it. <Esc> also closes a menu. <Enter> selects a menu item.

There is a conflict between using the <Alt> key to select menu items, and
using <Alt> key combinations for mappings. The 'winaltkeys' option tells Vim
what it should do with the <Alt> key.
   The default value "menu" is the smart choice: If the key combination is a
menu shortcut it can't be mapped. All other keys are available for mapping.
   The value "no" doesn't use any <Alt> keys for the menus. Thus you must use
the mouse for the menus, and all <Alt> keys can be mapped.
   The value "yes" means that Vim will use any <Alt> keys for the menus. Some
<Alt> key combinations may also do other things than selecting a menu.

31.4 Vim window position and size

To see the current Vim window position on the screen use:

```
:winpos
```

This will only work in the GUI. The output may look like this:

Window position: X 272, Y 103

The position is given in screen pixels. Now you can use the numbers to move
Vim somewhere else. For example, to move it to the left a hundred pixels:

```
:winpos 172 103
```

> **Note:**
> There may be a small offset between the reported position and where
> the window moves. This is because of the border around the window.
> This is added by the window manager.

You can use this command in your startup script to position the window at a
specific position.

The size of the Vim window is computed in characters. Thus this depends on
the size of the font being used. You can see the current size with this
command:

```
:set lines columns
```

To change the size set the 'lines' and/or 'columns' options to a new value:

```
:set lines=50
:set columns=80
```

Obtaining the size works in a terminal just like in the GUI. Setting the size
is not possible in most terminals.

You can start the X-Windows version of gvim with an argument to specify the
size and position of the window:

```
gvim -geometry {width}x{height}+{x-offset}+{y-offset}
```

{width} and {height} are in characters, {x-offset} and {y-offset} are in
pixels. Example:

```
gvim -geometry 80x25+100+300
```

31.5 Various

You can use gvim to edit an e-mail message. In your e-mail program you must
select gvim to be the editor for messages. When you try that, you will
see that it doesn't work: The mail program thinks that editing is finished,
while gvim is still running!
   What happens is that gvim disconnects from the shell it was started in.
That is fine when you start gvim in a terminal, so that you can do other work
in that terminal. But when you really want to wait for gvim to finish, you
must prevent it from disconnecting. The "-f" argument does this:

```
gvim -f file.txt
```

The "-f" stands for foreground. Now Vim will block the shell it was started
in until you finish editing and exit.

DELAYED START OF THE GUI

On Unix it's possible to first start Vim in a terminal. That's useful if you
do various tasks in the same shell. If you are editing a file and decide you
want to use the GUI after all, you can start it with:

```
:gui
```

Vim will open the GUI window and no longer use the terminal. You can continue
using the terminal for something else. The "-f" argument is used here to run
the GUI in the foreground. You can also use ":gui -f".

Next chapter: usr_32.txt The undo tree

Copyright: see manual-copyright

The undo tree

Vim provides multi-level undo. If you undo a few changes and then make a new change you create a branch in the undo tree. This text is about moving through the branches.

32.1 Undo up to a file write
32.2 Numbering changes
32.3 Jumping around the tree
32.4 Time travelling

Next chapter: usr_40.txt Make new commands
Previous chapter: usr_31.txt Exploiting the GUI
Table of contents: usr_toc.txt

Undo up to a file write

Sometimes you make several changes, and then discover you want to go back to when you have last written the file. You can do that with this command:

```
:earlier 1f
```

The "f" stands for "file" here.

You can repeat this command to go further back in the past. Or use a count different from 1 to go back faster.

If you go back too far, go forward again with:

```
:later 1f
```

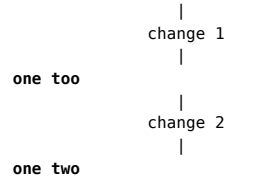
Note that these commands really work in time sequence. This matters if you made changes after undoing some changes. It's explained in the next section.

Also note that we are talking about text writes here. For writing the undo information in a file see undo-persistence.

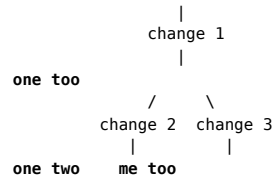
32.2 Numbering changes

In section 02.5 we only discussed one line of undo/redo. But it is also possible to branch off. This happens when you undo a few changes and then make a new change. The new changes become a branch in the undo tree.

Let's start with the text "one". The first change to make is to append " too". And then move to the first 'o' and change it into 'w'. We then have two changes, numbered 1 and 2, and three states of the text:



If we now undo one change, back to "one too", and change "one" to "me" we create a branch in the undo tree:



You can now use the u command to undo. If you do this twice you get to "one". Use CTRL-R to redo, and you will go to "one too". One more CTRL-R takes you to "me too". Thus undo and redo go up and down in the tree, using the branch that was last used.

What matters here is the order in which the changes are made. Undo and redo are not considered changes in this context. After each change you have a new state of the text.

Note that only the changes are numbered, the text shown in the tree above has no identifier. They are mostly referred to by the number of the change above it. But sometimes by the number of one of the changes below it, especially when moving up in the tree, so that you know which change was just undone.

32.3 Jumping around the tree

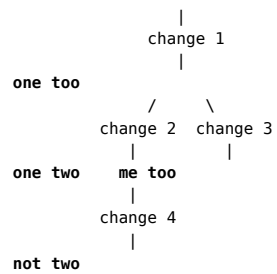
So how do you get to "one two" now? You can use this command:

```
:undo 2
```

The text is now "one two", you are below change 2. You can use the :undo

command to jump to below any change in the tree.

Now make another change: change "one" to "not":



Now you change your mind and want to go back to "me too". Use the g- command. This moves back in time. Thus it doesn't walk the tree upwards or downwards, but goes to the change made before.

You can repeat g- and you will see the text change:
me too
one two
one too
one

Use g+ to move forward in time:
one
one too
one two
me too
not two

Using :undo is useful if you know what change you want to jump to. g- and g+ are useful if you don't know exactly what the change number is.

You can type a count before g- and g+ to repeat them.

32.4 Time travelling

When you have been working on text for a while the tree grows to become big. Then you may want to go to the text of some minutes ago.

To see what branches there are in the undo tree use this command:

```
:undolist
```

```
number changes  time
             3       2  16 seconds ago
             4       3  5 seconds ago
```

Here you can see the number of the leaves in each branch and when the change was made. Assuming we are below change 4, at "not two", you can go back ten seconds with this command:

```
:earlier 10s
```

Depending on how much time you took for the changes you end up at a certain position in the tree. The :earlier command argument can be "m" for minutes, "h" for hours and "d" for days. To go all the way back use a big number:

```
:earlier 100d
```

To travel forward in time again use the :later command:

```
:later 1m
```

The arguments are "s", "m" and "h", just like with :earlier.

If you want even more details, or want to manipulate the information, you can use the undotree() function. To see what it returns:

```
:echo undotree()
```

Next chapter: usr_40.txt Make new commands

Copyright: see manual-copyright

Make new commands

Vim is an extensible editor. You can take a sequence of commands you use
often and turn it into a new command. Or redefine an existing command.
Autocommands make it possible to execute commands automatically.

40.1 Key mapping
40.2 Defining command-line commands
40.3 Autocommands

Next chapter: usr_41.txt Write a Vim script
Previous chapter: usr_32.txt The undo tree
Table of contents: usr_toc.txt

Key mapping

A simple mapping was explained in section 05.3. The principle is that one
sequence of key strokes is translated into another sequence of key strokes.
This is a simple, yet powerful mechanism.
   The simplest form is that one key is mapped to a sequence of keys. Since
the function keys, except <F1>, have no predefined meaning in Vim, these are
good choices to map. Example:

```
:map <F2> GoDate: <Esc>:read !date<CR>kJ
```

This shows how three modes are used. After going to the last line with "G",
the "o" command opens a new line and starts Insert mode. The text "Date: " is
inserted and <Esc> takes you out of insert mode.
   Notice the use of special keys inside <>. This is called angle bracket
notation. You type these as separate characters, not by pressing the key
itself. This makes the mappings better readable and you can copy and paste
the text without problems.
   The ":" character takes Vim to the command line. The ":read !date" command
reads the output from the "date" command and appends it below the current
line. The <CR> is required to execute the ":read" command.
   At this point of execution the text looks like this:

Date:
Fri Jun 15 12:54:34 CEST 2001

Now "kJ" moves the cursor up and joins the lines together.
   To decide which key or keys you use for mapping, see map-which-keys.

MAPPING AND MODES

The ":map" command defines remapping for keys in Normal mode. You can also
define mappings for other modes. For example, ":imap" applies to Insert mode.
You can use it to insert a date below the cursor:

```
:imap <F2> <CR>Date: <Esc>:read !date<CR>kJ
```

It looks a lot like the mapping for <F2> in Normal mode, only the start is
different. The <F2> mapping for Normal mode is still there. Thus you can map
the same key differently for each mode.
   Notice that, although this mapping starts in Insert mode, it ends in Normal
mode. If you want it to continue in Insert mode, append an "a" to the
mapping.

Here is an overview of map commands and in which mode they work:

```
:map            Normal, Visual and Operator-pending
:vmap           Visual
:nmap           Normal
:omap           Operator-pending
:map!           Insert and Command-line
:imap           Insert
:cmap           Command-line
```

Operator-pending mode is when you typed an operator character, such as "d" or
"y", and you are expected to type the motion command or a text object. Thus
when you type "dw", the "w" is entered in operator-pending mode.

Suppose that you want to define <F7> so that the command d<F7> deletes a C
program block (text enclosed in curly braces, {}). Similarly y<F7> would yank
the program block into the unnamed register. Therefore, what you need to do
is to define <F7> to select the current program block. You can do this with
the following command:

```
:omap <F7> a{
```

This causes <F7> to perform a select block "a{" in operator-pending mode, just
like you typed it. This mapping is useful if typing a { on your keyboard is a
bit difficult.

LISTING MAPPINGS

To see the currently defined mappings, use ":map" without arguments. Or one
of the variants that include the mode in which they work. The output could
look like this:

_g	:call MyGrep(1)<CR>
v <F2> :s/^/> /<CR>:noh<CR>``** **n <F2> :.,$s/^/> /<CR>:noh<CR>``	
<xHome>	<Home>
<xEnd>	<End>

The first column of the list shows in which mode the mapping is effective.
This is "n" for Normal mode, "i" for Insert mode, etc. A blank is used for a
mapping defined with ":map", thus effective in both Normal and Visual mode.
   One useful purpose of listing the mapping is to check if special keys in <>
form have been recognized (this only works when color is supported). For
example, when <Esc> is displayed in color, it stands for the escape character.
When it has the same color as the other text, it is five characters.

REMAPPING

The result of a mapping is inspected for other mappings in it. For example,
the mappings for <F2> above could be shortened to:

```
:map <F2> G<F3>
:imap <F2> <Esc><F3>
:map <F3>  oDate: <Esc>:read !date<CR>kJ
```

For Normal mode <F2> is mapped to go to the last line, and then behave like
<F3> was pressed. In Insert mode <F2> stops Insert mode with <Esc> and then
also uses <F3>. Then <F3> is mapped to do the actual work.

Suppose you hardly ever use Ex mode, and want to use the "Q" command to format
text (this was so in old versions of Vim). This mapping will do it:

```
:map Q gq
```

But, in rare cases you need to use Ex mode anyway. Let's map "gQ" to Q, so
that you can still go to Ex mode:

```
:map gQ Q
```

What happens now is that when you type "gQ" it is mapped to "Q". So far so
good. But then "Q" is mapped to "gq", thus typing "gQ" results in "gq", and
you don't get to Ex mode at all.
   To avoid keys to be mapped again, use the ":noremap" command:

```
:noremap gQ Q
```

Now Vim knows that the "Q" is not to be inspected for mappings that apply to
it. There is a similar command for every mode:

```
        :noremap        Normal, Visual and Operator-pending
        :vnoremap       Visual
        :nnoremap       Normal
        :onoremap       Operator-pending
        :noremap!       Insert and Command-line
        :inoremap       Insert
        :cnoremap       Command-line
```

RECURSIVE MAPPING

When a mapping triggers itself, it will run forever. This can be used to
repeat an action an unlimited number of times.
   For example, you have a list of files that contain a version number in the
first line. You edit these files with vim *.txt. You are now editing the
first file. Define this mapping:

```
:map ,, :s/5.1/5.2/<CR>:wnext<CR>,,
```

Now you type ",,". This triggers the mapping. It replaces "5.1" with "5.2"
in the first line. Then it does a ":wnext" to write the file and edit the
next one. The mapping ends in ",,". This triggers the same mapping again,
thus doing the substitution, etc.
   This continues until there is an error. In this case it could be a file
where the substitute command doesn't find a match for "5.1". You can then
make a change to insert "5.1" and continue by typing ",," again. Or the
":wnext" fails, because you are in the last file in the list.
   When a mapping runs into an error halfway, the rest of the mapping is
discarded. CTRL-C interrupts the mapping (CTRL-Break on MS-Windows).

DELETE A MAPPING

To remove a mapping use the ":unmap" command. Again, the mode the unmapping
applies to depends on the command used:

```
        :unmap          Normal, Visual and Operator-pending
        :vunmap         Visual
        :nunmap         Normal
        :ounmap         Operator-pending
```

```
        :unmap!         Insert and Command-line
        :iunmap         Insert
        :cunmap         Command-line
```

There is a trick to define a mapping that works in Normal and Operator-pending mode, but not in Visual mode. First define it for all three modes, then delete it for Visual mode:

```
:map <C-A> /---><CR>
:vunmap <C-A>
```

Notice that the five characters "<C-A>" stand for the single key CTRL-A.

To remove all mappings use the :mapclear command. You can guess the variations for different modes by now. Be careful with this command, it can't be undone.

SPECIAL CHARACTERS

The ":map" command can be followed by another command. A | character separates the two commands. This also means that a | character can't be used inside a map command. To include one, use <Bar> (five characters). Example:

```
:map <F8> :write <Bar> !checkin %:S<CR>
```

The same problem applies to the ":unmap" command, with the addition that you have to watch out for trailing white space. These two commands are different:

```
:unmap a | unmap b
:unmap a| unmap b
```

The first command tries to unmap "a ", with a trailing space.

When using a space inside a mapping, use <Space> (seven characters):

```
:map <Space> W
```

This makes the spacebar move a blank-separated word forward.

It is not possible to put a comment directly after a mapping, because the " character is considered to be part of the mapping. You can use |", this starts a new, empty command with a comment. Example:

```
:map <Space> W|     " Use spacebar to move forward a word
```

MAPPINGS AND ABBREVIATIONS

Abbreviations are a lot like Insert mode mappings. The arguments are handled in the same way. The main difference is the way they are triggered. An abbreviation is triggered by typing a non-word character after the word. A mapping is triggered when typing the last character.
   Another difference is that the characters you type for an abbreviation are inserted in the text while you type them. When the abbreviation is triggered these characters are deleted and replaced by what the abbreviation produces. When typing the characters for a mapping, nothing is inserted until you type the last character that triggers it. If the 'showcmd' option is set, the typed characters are displayed in the last line of the Vim window.
   An exception is when a mapping is ambiguous. Suppose you have done two mappings:

```
:imap aa foo
:imap aaa bar
```

Now, when you type "aa", Vim doesn't know if it should apply the first or the second mapping. It waits for another character to be typed. If it is an "a", the second mapping is applied and results in "bar". If it is a space, for example, the first mapping is applied, resulting in "foo", and then the space is inserted.

ADDITIONALLY...

The <script> keyword can be used to make a mapping local to a script. See :map-<script>.

The <buffer> keyword can be used to make a mapping local to a specific buffer. See :map-<buffer>

The <unique> keyword can be used to make defining a new mapping fail when it already exists. Otherwise a new mapping simply overwrites the old one. See :map-<unique>.

To make a key do nothing, map it to <Nop> (five characters). This will make the <F7> key do nothing at all:

```
:map <F7> <Nop>| map! <F7> <Nop>
```

There must be no space after <Nop>.

40.2 Defining command-line commands

The Vim editor enables you to define your own commands. You execute these
commands just like any other Command-line mode command.
   To define a command, use the ":command" command, as follows:

```
:command DeleteFirst 1delete
```

Now when you execute the command ":DeleteFirst" Vim executes ":1delete", which
deletes the first line.

> **Note:**
> User-defined commands must start with a capital letter. You cannot
> use ":Next". The underscore cannot be used! You can use digits, but
> this is discouraged.

To list the user-defined commands, execute the following command:

```
:command
```

Just like with the builtin commands, the user defined commands can be
abbreviated. You need to type just enough to distinguish the command from
another. Command line completion can be used to get the full name.

NUMBER OF ARGUMENTS

User-defined commands can take a series of arguments. The number of arguments
must be specified by the -nargs option. For instance, the example
:DeleteFirst command takes no arguments, so you could have defined it as
follows:

```
:command -nargs=0 DeleteFirst 1delete
```

However, because zero arguments is the default, you do not need to add
"-nargs=0". The other values of -nargs are as follows:

```
        -nargs=0        No arguments
        -nargs=1        One argument
        -nargs=*        Any number of arguments
        -nargs=?        Zero or one argument
        -nargs=+        One or more arguments
```

USING THE ARGUMENTS

Inside the command definition, the arguments are represented by the
<args> keyword. For example:

```
:command -nargs=+ Say :echo "<args>"
```

Now when you type

```
:Say Hello World
```

Vim echoes "Hello World". However, if you add a double quote, it won't work.
For example:

```
:Say he said "hello"
```

To get special characters turned into a string, properly escaped to use as an
expression, use "<q-args>":

```
:command -nargs=+ Say :echo <q-args>
```

Now the above ":Say" command will result in this to be executed:

```
:echo "he said \"hello\""
```

The <f-args> keyword contains the same information as the <args> keyword,
except in a format suitable for use as function call arguments. For example:

```
:command -nargs=* DoIt :call AFunction(<f-args>)
:DoIt a b c
```

Executes the following command:

```
:call AFunction("a", "b", "c")
```

LINE RANGE

Some commands take a range as their argument. To tell Vim that you are defining such a command, you need to specify a -range option. The values for this option are as follows:

```
        -range          Range is allowed; default is the current line.
        -range=%        Range is allowed; default is the whole file.
        -range={count}  Range is allowed; the last number in it is used as a
                        single number whose default is {count}.
```

When a range is specified, the keywords <line1> and <line2> get the values of the first and last line in the range. For example, the following command defines the SaveIt command, which writes out the specified range to the file "save_file":

```
:command -range=% SaveIt :<line1>,<line2>write! save_file
```

OTHER OPTIONS

Some of the other options and keywords are as follows:

```
        -count={number}         The command can take a count whose default is
                                {number}.  The resulting count can be used
                                through the <count> keyword.
        -bang                   You can use a !.  If present, using <bang>
                                will result in a !.
        -register               You can specify a register.  (The default is
                                the unnamed register.)
                                The register specification is available as
                                <reg> (a.k.a. <register>).
        -complete={type}        Type of command-line completion used.  See
                                :command-completion for the list of possible
                                values.
        -bar                    The command can be followed by | and another
                                command, or " and a comment.
        -buffer                 The command is only available for the current
                                buffer.
```

Finally, you have the <lt> keyword. It stands for the character <. Use this to escape the special meaning of the <> items mentioned.

REDEFINING AND DELETING

To redefine the same command use the ! argument:

```
:command -nargs=+ Say :echo "<args>"
:command! -nargs=+ Say :echo <q-args>
```

To delete a user command use ":delcommand". It takes a single argument, which is the name of the command. Example:

```
:delcommand SaveIt
```

To delete all the user commands:

```
:comclear
```

Careful, this can't be undone!

More details about all this in the reference manual: user-commands.

40.3 Autocommands

An autocommand is a command that is executed automatically in response to some event, such as a file being read or written or a buffer change. Through the use of autocommands you can train Vim to edit compressed files, for example. That is used in the gzip plugin.

   Autocommands are very powerful. Use them with care and they will help you avoid typing many commands. Use them carelessly and they will cause a lot of trouble.

Suppose you want to replace a datestamp on the end of a file every time it is written. First you define a function:

```
:function DateInsert()
:  $delete
:  read !date
:endfunction
```

You want this function to be called each time, just before a buffer is written to a file. This will make that happen:

```
:autocmd BufWritePre *  call DateInsert()
```

"BufWritePre" is the event for which this autocommand is triggered: Just

before (pre) writing a buffer to a file. The "*" is a pattern to match with
the file name. In this case it matches all files.
   With this command enabled, when you do a ":write", Vim checks for any
matching BufWritePre autocommands and executes them, and then it
performs the ":write".
   The general form of the :autocmd command is as follows:

```
:autocmd [group] {events} {file-pattern} [++nested] {command}
```

The [group] name is optional. It is used in managing and calling the commands
(more on this later). The {events} parameter is a list of events (comma
separated) that trigger the command.
   {file-pattern} is a filename, usually with wildcards. For example, using
"*.txt" makes the autocommand be used for all files whose name end in ".txt".
The optional [++nested] flag allows for nesting of autocommands (see below),
and finally, {command} is the command to be executed.

When adding an autocommand the already existing ones remain. To avoid adding
the autocommand several times you should use this form:

```
:augroup updateDate
:  autocmd!
:  autocmd BufWritePre *  call DateInsert()
:augroup END
```

This will delete any previously defined autocommand with :autocmd! before
defining the new one. Groups are explained later.

EVENTS

One of the most useful events is BufReadPost. It is triggered after a new
file is being edited. It is commonly used to set option values. For example,
you know that "*.gsm" files are GNU assembly language. To get the syntax file
right, define this autocommand:

```
:autocmd BufReadPost *.gsm  set filetype=asm
```

If Vim is able to detect the type of file, it will set the 'filetype' option
for you. This triggers the Filetype event. Use this to do something when a
certain type of file is edited. For example, to load a list of abbreviations
for text files:

```
:autocmd Filetype text  source ~/.config/nvim/abbrevs.vim
```

When starting to edit a new file, you could make Vim insert a skeleton:

```
:autocmd BufNewFile *.[ch]  0read ~/skeletons/skel.c
```

See autocmd-events for a complete list of events.

PATTERNS

The {file-pattern} argument can actually be a comma-separated list of file
patterns. For example: *.c,*.h matches files ending in ".c" and ".h".
   The usual file wildcards can be used. Here is a summary of the most often
used ones:

```
*               Match any character any number of times
?               Match any character once
[abc]           Match the character a, b or c
.               Matches a dot
a{b,c}          Matches "ab" and "ac"
```

When the pattern includes a slash (/) Vim will compare directory names.
Without the slash only the last part of a file name is used. For example,
"*.txt" matches "/home/biep/readme.txt". The pattern "/home/biep/*" would
also match it. But "home/foo/*.txt" wouldn't.
   When including a slash, Vim matches the pattern against both the full path
of the file ("/home/biep/readme.txt") and the relative path (e.g.,
"biep/readme.txt").

> **Note:**
> When working on a system that uses a backslash as file separator, such
> as MS-Windows, you still use forward slashes in autocommands. This
> makes it easier to write the pattern, since a backslash has a special
> meaning. It also makes the autocommands portable.

DELETING

To delete an autocommand, use the same command as what it was defined with,
but leave out the {command} at the end and use a !. Example:

```
:autocmd! FileWritePre *
```

This will delete all autocommands for the "FileWritePre" event that use the
"*" pattern.

LISTING

To list all the currently defined autocommands, use this:

```
:autocmd
```

The list can be very long, especially when filetype detection is used. To
list only part of the commands, specify the group, event and/or pattern. For
example, to list all BufNewFile autocommands:

```
:autocmd BufNewFile
```

To list all autocommands for the pattern "*.c":

```
:autocmd * *.c
```

Using "*" for the event will list all the events. To list all autocommands
for the cprograms group:

```
:autocmd cprograms
```

GROUPS

The {group} item, used when defining an autocommand, groups related
autocommands together. This can be used to delete all the autocommands in a
certain group, for example.
   When defining several autocommands for a certain group, use the ":augroup"
command. For example, let's define autocommands for C programs:

```
:augroup cprograms
:  autocmd BufReadPost *.c,*.h :set sw=4 sts=4
:  autocmd BufReadPost *.cpp   :set sw=3 sts=3
:augroup END
```

This will do the same as:

```
:autocmd cprograms BufReadPost *.c,*.h :set sw=4 sts=4
:autocmd cprograms BufReadPost *.cpp   :set sw=3 sts=3
```

To delete all autocommands in the "cprograms" group:

```
:autocmd! cprograms
```

NESTING

Generally, commands executed as the result of an autocommand event will not
trigger any new events. If you read a file in response to a FileChangedShell
event, it will not trigger the autocommands that would set the syntax, for
example. To make the events triggered, add the "++nested" flag:

```
:autocmd FileChangedShell * ++nested  edit
```

EXECUTING AUTOCOMMANDS

It is possible to trigger an autocommand by pretending an event has occurred.
This is useful to have one autocommand trigger another one. Example:

```
:autocmd BufReadPost *.new  execute "doautocmd BufReadPost " .. expand("<afile>:r")
```

This defines an autocommand that is triggered when a new file has been edited.
The file name must end in ".new". The ":execute" command uses expression
evaluation to form a new command and execute it. When editing the file
"tryout.c.new" the executed command will be:

```
:doautocmd BufReadPost tryout.c
```

The expand() function takes the "<afile>" argument, which stands for the file
name the autocommand was executed for, and takes the root of the file name
with ":r".

":doautocmd" executes on the current buffer. The ":doautoall" command works
like "doautocmd" except it executes on all the buffers.

USING NORMAL MODE COMMANDS

The commands executed by an autocommand are Command-line commands. If you
want to use a Normal mode command, the ":normal" command can be used.
Example:

```
:autocmd BufReadPost *.log normal G
```

This will make the cursor jump to the last line of *.log files when you start to edit it.
   Using the ":normal" command is a bit tricky. First of all, make sure its argument is a complete command, including all the arguments. When you use "i" to go to Insert mode, there must also be a <Esc> to leave Insert mode again. If you use a "/" to start a search pattern, there must be a <CR> to execute it.
   The ":normal" command uses all the text after it as commands. Thus there can be no | and another command following. To work around this, put the ":normal" command inside an ":execute" command. This also makes it possible to pass unprintable characters in a convenient way. Example:

```
:autocmd BufReadPost *.chg execute "normal ONew entry:\<Esc>" |
        \ 1read !date
```

This also shows the use of a backslash to break a long command into more lines. This can be used in Vim scripts (not at the command line).

When you want the autocommand do something complicated, which involves jumping around in the file and then returning to the original position, you may want to restore the view on the file. See restore-position for an example.

IGNORING EVENTS

At times, you will not want to trigger an autocommand. The 'eventignore' option contains a list of events that will be totally ignored. For example, the following causes events for entering and leaving a window to be ignored:

```
:set eventignore=WinEnter,WinLeave
```

To ignore all events, use the following command:

```
:set eventignore=all
```

To set it back to the normal behavior, make 'eventignore' empty:

```
:set eventignore=
```

Next chapter: usr_41.txt Write a Vim script

Copyright: see manual-copyright

VIM USER MANUAL by Bram Moolenaar

Write a Vim script

The Vim script language is used for the startup vimrc file, syntax files, and many other things. This chapter explains the items that can be used in a Vim script. There are a lot of them, thus this is a long chapter.

41.1 Introduction
41.2 Variables
41.3 Expressions
41.4 Conditionals
41.5 Executing an expression
41.6 Using functions
41.7 Defining a function
41.8 Lists and Dictionaries
41.9 Exceptions
41.10 Various remarks
41.11 Writing a plugin
41.12 Writing a filetype plugin
41.13 Writing a compiler plugin
41.14 Writing a plugin that loads quickly
41.15 Writing library scripts
41.16 Distributing Vim scripts

Next chapter: usr_42.txt Add new menus
Previous chapter: usr_40.txt Make new commands
Table of contents: usr_toc.txt

Introduction vimscript-intro

Your first experience with Vim scripts is the vimrc file. Vim reads it when it starts up and executes the commands. You can set options to values you prefer. And you can use any colon command in it (commands that start with a ":"; these are sometimes referred to as Ex commands or command-line commands).
   Syntax files are also Vim scripts. As are files that set options for a specific file type. A complicated macro can be defined by a separate Vim script file. You can think of other uses yourself.

> If you are familiar with Python, you can find a comparison between Python and Vim script here, with pointers to other documents:
> https://gist.github.com/yegappan/16d964a37ead0979b05e655aa036cad0
> And if you are familiar with JavaScript:
> https://w0rp.com/blog/post/vim-script-for-the-javascripter/

Let's start with a simple example:

```
:let i = 1
:while i < 5
:  echo "count is" i
:  let i += 1
:endwhile
```

> **Note:**
> The ":" characters are not really needed here. You only need to use them when you type a command. In a Vim script file they can be left out. We will use them here anyway to make clear these are colon commands and make them stand out from Normal mode commands.
> **Note:**
> You can try out the examples by yanking the lines from the text here and executing them with :@"

The output of the example code is:

```
count is 1
count is 2
count is 3
count is 4
```

In the first line the ":let" command assigns a value to a variable. The generic form is:

```
:let {variable} = {expression}
```

In this case the variable name is "i" and the expression is a simple value, the number one.
   The ":while" command starts a loop. The generic form is:

```
:while {condition}
:  {statements}
:endwhile
```

The statements until the matching ":endwhile" are executed for as long as the condition is true. The condition used here is the expression "i < 5". This is true when the variable i is smaller than five.

> **Note:**
> If you happen to write a while loop that keeps on running, you can interrupt it by pressing CTRL-C (CTRL-Break on MS-Windows).

The ":echo" command prints its arguments. In this case the string "count is" and the value of the variable i. Since i is one, this will print:

```
count is 1
```

Then there is the ":let i += 1" command. This does the same thing as
":let i = i + 1". This adds one to the variable i and assigns the new value
to the same variable.

The example was given to explain the commands, but would you really want to
make such a loop, it can be written much more compact:

```
:for i in range(1, 4)
:  echo "count is" i
:endfor
```

We won't explain how :for and range() work until later. Follow the links
if you are impatient.

FOUR KINDS OF NUMBERS

Numbers can be decimal, hexadecimal, octal or binary.

A hexadecimal number starts with "0x" or "0X". For example "0x1f" is decimal

31.

An octal number starts with "0o", "0O" or a zero and another digit. "0o17" is
decimal 15.

A binary number starts with "0b" or "0B". For example "0b101" is decimal 5.

A decimal number is just digits. Careful: don't put a zero before a decimal
number, it will be interpreted as an octal number!

The ":echo" command always prints decimal numbers. Example:

```
:echo 0x7f 0o36
```

127 30

A number is made negative with a minus sign. This also works for hexadecimal,
octal and binary numbers. A minus sign is also used for subtraction. Compare
this with the previous example:

```
:echo 0x7f -0o36
```

97

White space in an expression is ignored. However, it's recommended to use it
for separating items, to make the expression easier to read. For example, to
avoid the confusion with a negative number above, put a space between the
minus sign and the following number:

```
:echo 0x7f - 0o36
```

41.2 Variables

A variable name consists of ASCII letters, digits and the underscore. It
cannot start with a digit. Valid variable names are:

```
counter
_aap3
very_long_variable_name_with_underscores
FuncLength
LENGTH
```

Invalid names are "foo.bar" and "6var".
   These variables are global. To see a list of currently defined variables
use this command:

```
:let
```

You can use global variables everywhere. This also means that when the
variable "count" is used in one script file, it might also be used in another
file. This leads to confusion at least, and real problems at worst. To avoid
this, you can use a variable local to a script file by prepending "s:". For
example, one script contains this code:

```
:let s:count = 1
:while s:count < 5
:  source other.vim
:  let s:count += 1
:endwhile
```

Since "s:count" is local to this script, you can be sure that sourcing the
"other.vim" script will not change this variable. If "other.vim" also uses an
"s:count" variable, it will be a different copy, local to that script. More
about script-local variables here: script-variable.

There are more kinds of variables, see internal-variables. The most often
used ones are:

```
b:name          variable local to a buffer
w:name          variable local to a window
```

```
        g:name          global variable (also in a function)
        v:name          variable predefined by Vim
```

DELETING VARIABLES

Variables take up memory and show up in the output of the ":let" command. To delete a variable use the ":unlet" command. Example:

```
:unlet s:count
```

This deletes the script-local variable "s:count" to free up the memory it uses. If you are not sure if the variable exists, and don't want an error message when it doesn't, append !:

```
:unlet! s:count
```

When a script has been processed to the end, the local variables declared there will not be deleted. Functions defined in the script can use them. Example:

```
        :if !exists("s:call_count")
        :  let s:call_count = 0
        :endif
        :let s:call_count = s:call_count + 1
        :echo "called" s:call_count "times"
```

The "exists()" function checks if a variable has already been defined. Its argument is the name of the variable you want to check. Not the variable itself! If you would do this:

```
:if !exists(s:call_count)
```

Then the value of s:call_count will be used as the name of the variable that exists() checks. That's not what you want.

The exclamation mark ! negates a value. When the value was true, it becomes false. When it was false, it becomes true. You can read it as "not". Thus "if !exists()" can be read as "if not exists()".

What Vim calls true is anything that is not zero. Zero is false.

Note:
Vim automatically converts a string to a number when it is looking for a number. When using a string that doesn't start with a digit the resulting number is zero. Thus look out for this:

```
:if "true"
```

The "true" will be interpreted as a zero, thus as false!

STRING VARIABLES AND CONSTANTS

So far only numbers were used for the variable value. Strings can be used as well. Numbers and strings are the basic types of variables that Vim supports. The type is dynamic, it is set each time when assigning a value to the variable with ":let". More about types in 41.8.

To assign a string value to a variable, you need to use a string constant. There are two types of these. First the string in double quotes:

```
:let name = "peter"
:echo name
```

peter

If you want to include a double quote inside the string, put a backslash in front of it:

```
:let name = "\"peter\""
:echo name
```

"peter"

To avoid the need for a backslash, you can use a string in single quotes:

```
:let name = '"peter"'
:echo name
```

"peter"

Inside a single-quote string all the characters are as they are. Only the single quote itself is special: you need to use two to get one. A backslash is taken literally, thus you can't use it to change the meaning of the character after it.

In double-quote strings it is possible to use special characters. Here are a few useful ones:

```
        \t              <Tab>
        \n              <NL>, line break
        \r              <CR>, <Enter>
        \e              <Esc>
        \b              <BS>, backspace
        \"              "
        \\              \, backslash
```

```
        \<Esc>          <Esc>
        \<C-W>          CTRL-W
```

The last two are just examples. The "\<name>" form can be used to include the special key "name".

   See expr-quote for the full list of special items in a string.

41.3 Expressions

Vim has a rich, yet simple way to handle expressions. You can read the definition here: expression-syntax. Here we will show the most common items.

   The numbers, strings and variables mentioned above are expressions by themselves. Thus everywhere an expression is expected, you can use a number, string or variable. Other basic items in an expression are:

```
        $NAME           environment variable
        &name           option
        @r              register
```

Examples:

```
:echo "The value of 'tabstop' is" &ts
:echo "Your home directory is" $HOME
:if @a > 5
```

The &name form can be used to save an option value, set it to a new value, do something and restore the old value. Example:

```
:let save_ic = &ic
:set noic
:/The Start/,$delete
:let &ic = save_ic
```

This makes sure the "The Start" pattern is used with the 'ignorecase' option off. Still, it keeps the value that the user had set. (Another way to do this would be to add "\C" to the pattern, see /\C.)

MATHEMATICS

It becomes more interesting if we combine these basic items. Let's start with mathematics on numbers:

```
        a + b           add
        a - b           subtract
        a * b           multiply
        a / b           divide
        a % b           modulo
```

The usual precedence is used. Example:

```
:echo 10 + 5 * 2
```

20

Grouping is done with parentheses. No surprises here. Example:

```
:echo (10 + 5) * 2
```

30

Strings can be concatenated with ".." (see expr6). Example:

```
:echo "foo" .. "bar"
```

foobar

When the ":echo" command gets multiple arguments, it separates them with a space. In the example the argument is a single expression, thus no space is inserted.

Borrowed from the C language is the conditional expression:

```
        a ? b : c
```

If "a" evaluates to true "b" is used, otherwise "c" is used. Example:

```
:let i = 4
:echo i > 5 ? "i is big" : "i is small"
```

i is small

The three parts of the constructs are always evaluated first, thus you could see it work as:

```
        (a) ? (b) : (c)
```

41.4 Conditionals

The ":if" commands executes the following statements, until the matching ":endif", only when a condition is met. The generic form is:

```
        :if {condition}
           {statements}
        :endif
```

Only when the expression {condition} evaluates to true (non-zero) will the
{statements} be executed. These must still be valid commands. If they
contain garbage, Vim won't be able to find the ":endif".
   You can also use ":else". The generic form for this is:

```
:if {condition}
   {statements}
:else
   {statements}
:endif
```

The second {statements} is only executed if the first one isn't.
   Finally, there is ":elseif":

```
:if {condition}
   {statements}
:elseif {condition}
   {statements}
:endif
```

This works just like using ":else" and then "if", but without the need for an
extra ":endif".
   A useful example for your vimrc file is checking the 'term' option and
doing something depending upon its value:

```
:if &term == "xterm"
:  " Do stuff for xterm
:elseif &term == "vt100"
:  " Do stuff for a vt100 terminal
:else
:  " Do something for other terminals
:endif
```

LOGIC OPERATIONS

We already used some of them in the examples. These are the most often used
ones:

```
a == b          equal to
a != b          not equal to
a >  b          greater than
a >= b          greater than or equal to
a <  b          less than
a <= b          less than or equal to
```

The result is one if the condition is met and zero otherwise. An example:

```
:if v:version >= 700
:  echo "congratulations"
:else
:  echo "you are using an old version, upgrade!"
:endif
```

Here "v:version" is a variable defined by Vim, which has the value of the Vim
version. 600 is for version 6.0. Version 6.1 has the value 601. This is
very useful to write a script that works with multiple versions of Vim.
v:version

The logic operators work both for numbers and strings. When comparing two
strings, the mathematical difference is used. This compares byte values,
which may not be right for some languages.
   When comparing a string with a number, the string is first converted to a
number. This is a bit tricky, because when a string doesn't look like a
number, the number zero is used. Example:

```
:if 0 == "one"
:  echo "yes"
:endif
```

This will echo "yes", because "one" doesn't look like a number, thus it is
converted to the number zero.

For strings there are two more items:

```
a =~ b          matches with
a !~ b          does not match with
```

The left item "a" is used as a string. The right item "b" is used as a
pattern, like what's used for searching. Example:

```
:if str =~ " "
:  echo "str contains a space"
:endif
:if str !~ '\.$'
:  echo "str does not end in a full stop"
:endif
```

Notice the use of a single-quote string for the pattern. This is useful,
because backslashes would need to be doubled in a double-quote string and
patterns tend to contain many backslashes.

The 'ignorecase' option is used when comparing strings. When you don't want
that, append "#" to match case and "?" to ignore case. Thus "==?" compares
two strings to be equal while ignoring case. And "!~#" checks if a pattern
doesn't match, also checking the case of letters. For the full table see
expr-==.

MORE LOOPING

The ":while" command was already mentioned. Two more statements can be used
in between the ":while" and the ":endwhile":

:continue	Jump back to the start of the while loop; the loop continues.
:break	Jump forward to the ":endwhile"; the loop is discontinued.

Example:

```
:while counter < 40
:  call do_something()
:  if skip_flag
:    continue
:  endif
:  if finished_flag
:    break
:  endif
:  sleep 50m
:endwhile
```

The ":sleep" command makes Vim take a nap. The "50m" specifies fifty
milliseconds. Another example is ":sleep 4", which sleeps for four seconds.

Even more looping can be done with the ":for" command, see below in 41.8.

41.5 Executing an expression

So far the commands in the script were executed by Vim directly. The
":execute" command allows executing the result of an expression. This is a
very powerful way to build commands and execute them.
   An example is to jump to a tag, which is contained in a variable:

```
:execute "tag " .. tag_name
```

The ".." is used to concatenate the string "tag " with the value of variable
"tag_name". Suppose "tag_name" has the value "get_cmd", then the command that
will be executed is:

```
:tag get_cmd
```

The ":execute" command can only execute colon commands. The ":normal" command
executes Normal mode commands. However, its argument is not an expression but
the literal command characters. Example:

```
:normal gg=G
```

This jumps to the first line and formats all lines with the "=" operator.
   To make ":normal" work with an expression, combine ":execute" with it.
Example:

```
:execute "normal " .. normal_commands
```

The variable "normal_commands" must contain the Normal mode commands.
   Make sure that the argument for ":normal" is a complete command. Otherwise
Vim will run into the end of the argument and abort the command. For example,
if you start Insert mode, you must leave Insert mode as well. This works:

```
:execute "normal Inew text \<Esc>"
```

This inserts "new text " in the current line. Notice the use of the special
key "\<Esc>". This avoids having to enter a real <Esc> character in your
script.

If you don't want to execute a string but evaluate it to get its expression
value, you can use the eval() function:

```
:let optname = "path"
:let optval = eval('&' .. optname)
```

A "&" character is prepended to "path", thus the argument to eval() is
"&path". The result will then be the value of the 'path' option.
   The same thing can be done with:

```
:exe 'let optval = &' .. optname
```

41.6 Using functions

Vim defines many functions and provides a large amount of functionality that way. A few examples will be given in this section. You can find the whole list below: function-list.

A function is called with the ":call" command. The parameters are passed in between parentheses separated by commas. Example:

```
:call search("Date: ", "W")
```

This calls the search() function, with arguments "Date: " and "W". The search() function uses its first argument as a search pattern and the second one as flags. The "W" flag means the search doesn't wrap around the end of the file.

A function can be called in an expression. Example:

```
:let line = getline(".")
:let repl = substitute(line, '\a', "*", "g")
:call setline(".", repl)
```

The getline() function obtains a line from the current buffer. Its argument is a specification of the line number. In this case "." is used, which means the line where the cursor is.

The substitute() function does something similar to the ":substitute" command. The first argument is the string on which to perform the substitution. The second argument is the pattern, the third the replacement string. Finally, the last arguments are the flags.

The setline() function sets the line, specified by the first argument, to a new string, the second argument. In this example the line under the cursor is replaced with the result of the substitute(). Thus the effect of the three statements is equal to:

```
:substitute/\a/*/g
```

Using the functions becomes interesting when you do more work before and after the substitute() call.

FUNCTIONS function-list

There are many functions. We will mention them here, grouped by what they are used for. You can find an alphabetical list here: vimscript-functions.
Use CTRL-] on the function name to jump to detailed help on it.

String manipulation: string-functions

nr2char()	get a character by its number value
list2str()	get a character string from a list of numbers
char2nr()	get number value of a character
str2list()	get list of numbers from a string
str2nr()	convert a string to a Number
str2float()	convert a string to a Float
printf()	format a string according to "%" items
escape()	escape characters in a string with a '\'
shellescape()	escape a string for use with a shell command
fnameescape()	escape a file name for use with a Vim command
tr()	translate characters from one set to another
strtrans()	translate a string to make it printable
keytrans()	translate internal keycodes to a form that can be used by :map
tolower()	turn a string to lowercase
toupper()	turn a string to uppercase
charclass()	class of a character
match()	position where a pattern matches in a string
matchbufline()	all the matches of a pattern in a buffer
matchend()	position where a pattern match ends in a string
matchfuzzy()	fuzzy matches a string in a list of strings
matchfuzzypos()	fuzzy matches a string in a list of strings
matchstr()	match of a pattern in a string
matchstrlist()	all the matches of a pattern in a List of strings
matchstrpos()	match and positions of a pattern in a string
matchlist()	like matchstr() and also return submatches
stridx()	first index of a short string in a long string
strridx()	last index of a short string in a long string
strlen()	length of a string in bytes
strcharlen()	length of a string in characters
strchars()	number of characters in a string
strutf16len()	number of UTF-16 code units in a string
strwidth()	size of string when displayed
strdisplaywidth()	size of string when displayed, deals with tabs
setcellwidths()	set character cell width overrides
getcellwidths()	get character cell width overrides
reverse()	reverse the order of characters in a string
substitute()	substitute a pattern match with a string
submatch()	get a specific match in ":s" and substitute()

strpart()	get part of a string using byte index
strcharpart()	get part of a string using char index
slice()	take a slice of a string, using char index in Vim9 script
strgetchar()	get character from a string using char index
expand()	expand special keywords
expandcmd()	expand a command like done for :edit
iconv()	convert text from one encoding to another
byteidx()	byte index of a character in a string
byteidxcomp()	like byteidx() but count composing characters
charidx()	character index of a byte in a string
utf16idx()	UTF-16 index of a byte in a string
repeat()	repeat a string multiple times
eval()	evaluate a string expression
execute()	execute an Ex command and get the output
win_execute()	like execute() but in a specified window
trim()	trim characters from a string
gettext()	lookup message translation
items()	get List of String index-character pairs

List manipulation: list-functions

get()	get an item without error for wrong index
len()	number of items in a List
empty()	check if List is empty
insert()	insert an item somewhere in a List
add()	append an item to a List
extend()	append a List to a List
extendnew()	make a new List and append items
remove()	remove one or more items from a List
copy()	make a shallow copy of a List
deepcopy()	make a full copy of a List
filter()	remove selected items from a List
map()	change each List item
mapnew()	make a new List with changed items
foreach()	apply function to List items
reduce()	reduce a List to a value
slice()	take a slice of a List
sort()	sort a List
reverse()	reverse the order of items in a List
uniq()	remove copies of repeated adjacent items
split()	split a String into a List
join()	join List items into a String
range()	return a List with a sequence of numbers
string()	String representation of a List
call()	call a function with List as arguments
index()	index of a value in a List
indexof()	index in a List where an expression is true
max()	maximum value in a List
min()	minimum value in a List
count()	count number of times a value appears in a List
repeat()	repeat a List multiple times
flatten()	flatten a List
flattennew()	flatten a copy of a List
items()	get List of List index-value pairs

Dictionary manipulation: dict-functions

get()	get an entry without an error for a wrong key
len()	number of entries in a Dictionary
has_key()	check whether a key appears in a Dictionary
empty()	check if Dictionary is empty
remove()	remove an entry from a Dictionary
extend()	add entries from one Dictionary to another
extendnew()	make a new Dictionary and append items
filter()	remove selected entries from a Dictionary
map()	change each Dictionary entry
mapnew()	make a new Dictionary with changed items
foreach()	apply function to Dictionary items
keys()	get List of Dictionary keys
values()	get List of Dictionary values
items()	get List of Dictionary key-value pairs
copy()	make a shallow copy of a Dictionary
deepcopy()	make a full copy of a Dictionary
string()	String representation of a Dictionary
max()	maximum value in a Dictionary
min()	minimum value in a Dictionary
count()	count number of times a value appears

Floating point computation: float-functions

float2nr()	convert Float to Number
abs()	absolute value (also works for Number)
round()	round off
ceil()	round up
floor()	round down
trunc()	remove value after decimal point
fmod()	remainder of division
exp()	exponential
log()	natural logarithm (logarithm to base e)

log10() logarithm to base 10
pow() value of x to the exponent y
sqrt() square root
sin() sine
cos() cosine
tan() tangent
asin() arc sine
acos() arc cosine
atan() arc tangent
atan2() arc tangent
sinh() hyperbolic sine
cosh() hyperbolic cosine
tanh() hyperbolic tangent
isinf() check for infinity
isnan() check for not a number

Blob manipulation: blob-functions

blob2list() get a list of numbers from a blob
list2blob() get a blob from a list of numbers
reverse() reverse the order of numbers in a blob
index() index of a value in a Blob
indexof() index in a Blob where an expression is true
items() get List of Blob index-value pairs

Other computation: bitwise-function

and() bitwise AND
invert() bitwise invert
or() bitwise OR
xor() bitwise XOR
sha256() SHA-256 hash
rand() get a pseudo-random number
srand() initialize seed used by rand()

Variables: var-functions

type() type of a variable
islocked() check if a variable is locked
funcref() get a Funcref for a function reference
function() get a Funcref for a function name
getbufvar() get a variable value from a specific buffer
setbufvar() set a variable in a specific buffer
getwinvar() get a variable from specific window
gettabvar() get a variable from specific tab page
gettabwinvar() get a variable from specific window & tab page
setwinvar() set a variable in a specific window
settabvar() set a variable in a specific tab page
settabwinvar() set a variable in a specific window & tab page
garbagecollect() possibly free memory

Cursor and mark position: cursor-functions mark-functions

col() column number of the cursor or a mark
virtcol() screen column of the cursor or a mark
line() line number of the cursor or mark
wincol() window column number of the cursor
winline() window line number of the cursor
cursor() position the cursor at a line/column
screencol() get screen column of the cursor
screenrow() get screen row of the cursor
screenpos() screen row and col of a text character
virtcol2col() byte index of a text character on screen
getcurpos() get position of the cursor
getpos() get position of cursor, mark, etc.
setpos() set position of cursor, mark, etc.
getmarklist() list of global/local marks
byte2line() get line number at a specific byte count
line2byte() byte count at a specific line
diff_filler() get the number of filler lines above a line
screenattr() get attribute at a screen line/row
screenchar() get character code at a screen line/row
screenchars() get character codes at a screen line/row
screenstring() get string of characters at a screen line/row
charcol() character number of the cursor or a mark
getcharpos() get character position of cursor, mark, etc.
setcharpos() set character position of cursor, mark, etc.
getcursorcharpos() get character position of the cursor
setcursorcharpos() set character position of the cursor

Working with text in the current buffer: text-functions

getline() get a line or list of lines from the buffer
getregion() get a region of text from the buffer
getregionpos() get a list of positions for a region
setline() replace a line in the buffer
append() append line or list of lines in the buffer
indent() indent of a specific line
cindent() indent according to C indenting
lispindent() indent according to Lisp indenting
nextnonblank() find next non-blank line
prevnonblank() find previous non-blank line
search() find a match for a pattern

searchpos() find a match for a pattern
searchcount() get number of matches before/after the cursor
searchpair() find the other end of a start/skip/end
searchpairpos() find the other end of a start/skip/end
searchdecl() search for the declaration of a name
getcharsearch() return character search information
setcharsearch() set character search information

Working with text in another buffer:
getbufline() get a list of lines from the specified buffer
getbufoneline() get a one line from the specified buffer
setbufline() replace a line in the specified buffer
appendbufline() append a list of lines in the specified buffer
deletebufline() delete lines from a specified buffer

system-functions file-functions
System functions and manipulation of files:
glob() expand wildcards
globpath() expand wildcards in a number of directories
glob2regpat() convert a glob pattern into a search pattern
findfile() find a file in a list of directories
finddir() find a directory in a list of directories
resolve() find out where a shortcut points to
fnamemodify() modify a file name
pathshorten() shorten directory names in a path
simplify() simplify a path without changing its meaning
executable() check if an executable program exists
exepath() full path of an executable program
filereadable() check if a file can be read
filewritable() check if a file can be written to
getfperm() get the permissions of a file
setfperm() set the permissions of a file
getftype() get the kind of a file
isabsolutepath() check if a path is absolute
isdirectory() check if a directory exists
getfsize() get the size of a file
getcwd() get the current working directory
haslocaldir() check if current window used :lcd or :tcd
tempname() get the name of a temporary file
mkdir() create a new directory
chdir() change current working directory
delete() delete a file
rename() rename a file
system() get the result of a shell command as a string
systemlist() get the result of a shell command as a list
environ() get all environment variables
getenv() get one environment variable
setenv() set an environment variable
hostname() name of the system
readfile() read a file into a List of lines
readblob() read a file into a Blob
readdir() get a List of file names in a directory
writefile() write a List of lines or Blob into a file
filecopy() copy a file {from} to {to}

Date and Time: date-functions time-functions
getftime() get last modification time of a file
localtime() get current time in seconds
strftime() convert time to a string
strptime() convert a date/time string to time
reltime() get the current or elapsed time accurately
reltimestr() convert reltime() result to a string
reltimefloat() convert reltime() result to a Float

buffer-functions window-functions arg-functions
Buffers, windows and the argument list:
argc() number of entries in the argument list
argidx() current position in the argument list
arglistid() get id of the argument list
argv() get one entry from the argument list
bufadd() add a file to the list of buffers
bufexists() check if a buffer exists
buflisted() check if a buffer exists and is listed
bufload() ensure a buffer is loaded
bufloaded() check if a buffer exists and is loaded
bufname() get the name of a specific buffer
bufnr() get the buffer number of a specific buffer
tabpagebuflist() return List of buffers in a tab page
tabpagenr() get the number of a tab page
tabpagewinnr() like winnr() for a specified tab page
winnr() get the window number for the current window
bufwinid() get the window ID of a specific buffer
bufwinnr() get the window number of a specific buffer
winbufnr() get the buffer number of a specific window
win_findbuf() find windows containing a buffer
win_getid() get window ID of a window
win_gettype() get type of window
win_gotoid() go to window with ID
win_id2tabwin() get tab and window nr from window ID
win_id2win() get window nr from window ID
win_move_separator() move window vertical separator
win_move_statusline() move window status line

win_splitmove() move window to a split of another window
getbufinfo() get a list with buffer information
gettabinfo() get a list with tab page information
getwininfo() get a list with window information
getchangelist() get a list of change list entries
getjumplist() get a list of jump list entries
swapfilelist() list of existing swap files in 'directory'
swapinfo() information about a swap file
swapname() get the swap file path of a buffer

Command line: command-line-functions

getcmdcomplpat() get completion pattern of the current command line
getcmdcompltype() get the type of the current command line completion
getcmdline() get the current command line input
getcmdprompt() get the current command line prompt
getcmdpos() get position of the cursor in the command line
getcmdscreenpos() get screen position of the cursor in the command line
setcmdline() set the current command line
setcmdpos() set position of the cursor in the command line
getcmdtype() return the current command-line type
getcmdwintype() return the current command-line window type
getcompletion() list of command-line completion matches
getcompletiontype() get the type of the command-line completion for specified string
fullcommand() get full command name
cmdcomplete_info() get command-line completion information

Quickfix and location lists: quickfix-functions

getqflist() list of quickfix errors
setqflist() modify a quickfix list
getloclist() list of location list items
setloclist() modify a location list

Insert mode completion: completion-functions

complete() set found matches
complete_add() add to found matches
complete_check() check if completion should be aborted
complete_info() get current completion information
preinserted() check if text is inserted after cursor
pumvisible() check if the popup menu is displayed
pum_getpos() position and size of popup menu if visible

Folding: folding-functions

foldclosed() check for a closed fold at a specific line
foldclosedend() like foldclosed() but return the last line
foldlevel() check for the fold level at a specific line
foldtext() generate the line displayed for a closed fold
foldtextresult() get the text displayed for a closed fold

Syntax and highlighting: syntax-functions highlighting-functions
clearmatches() clear all matches defined by matchadd() and the :match commands
getmatches() get all matches defined by matchadd() and the :match commands
hlexists() check if a highlight group exists
hlID() get ID of a highlight group
synID() get syntax ID at a specific position
synIDattr() get a specific attribute of a syntax ID
synIDtrans() get translated syntax ID
synstack() get list of syntax IDs at a specific position
synconcealed() get info about (syntax) concealing
diff_hlID() get highlight ID for diff mode at a position
matchadd() define a pattern to highlight (a "match")
matchaddpos() define a list of positions to highlight
matcharg() get info about :match arguments
matchdelete() delete a match defined by matchadd() or a :match command
setmatches() restore a list of matches saved by getmatches()

Spelling: spell-functions

spellbadword() locate badly spelled word at or after cursor
spellsuggest() return suggested spelling corrections
soundfold() return the sound-a-like equivalent of a word

History: history-functions

histadd() add an item to a history
histdel() delete an item from a history
histget() get an item from a history
histnr() get highest index of a history list

Interactive: interactive-functions

browse() put up a file requester
browsedir() put up a directory requester

confirm() let the user make a choice
getchar() get a character from the user
getcharstr() get a character from the user as a string
getcharmod() get modifiers for the last typed character
getmousepos() get last known mouse position
feedkeys() put characters in the typeahead queue
input() get a line from the user
inputlist() let the user pick an entry from a list
inputsecret() get a line from the user without showing it
inputdialog() get a line from the user in a dialog
inputsave() save and clear typeahead
inputrestore() restore typeahead

GUI: gui-functions

getfontname() get name of current font being used
getwinpos() position of the Vim window
getwinposx() X position of the Vim window
getwinposy() Y position of the Vim window
balloon_show() set the balloon content
balloon_split() split a message for a balloon
balloon_gettext() get the text in the balloon

Vim server: server-functions

serverlist() return the list of server names
remote_startserver() run a server
remote_send() send command characters to a Vim server
remote_expr() evaluate an expression in a Vim server
server2client() send a reply to a client of a Vim server
remote_peek() check if there is a reply from a Vim server
remote_read() read a reply from a Vim server
foreground() move the Vim window to the foreground
remote_foreground() move the Vim server window to the foreground

Window size and position: window-size-functions

winheight() get height of a specific window
winwidth() get width of a specific window
win_screenpos() get screen position of a window
winlayout() get layout of windows in a tab page
winrestcmd() return command to restore window sizes
winsaveview() get view of current window
winrestview() restore saved view of current window

Mappings and Menus: mapping-functions

digraph_get() get digraph
digraph_getlist() get all digraphs
digraph_set() register digraph
digraph_setlist() register multiple digraphs
hasmapto() check if a mapping exists
mapcheck() check if a matching mapping exists
maparg() get rhs of a mapping
maplist() get list of all mappings
mapset() restore a mapping
menu_info() get information about a menu item
wildmenumode() check if the wildmode is active
wildtrigger() start wildcard expansion

Signs: sign-functions

sign_define() define or update a sign
sign_getdefined() get a list of defined signs
sign_getplaced() get a list of placed signs
sign_jump() jump to a sign
sign_place() place a sign
sign_placelist() place a list of signs
sign_undefine() undefine a sign
sign_unplace() unplace a sign
sign_unplacelist() unplace a list of signs

Testing: test-functions

assert_equal() assert that two expressions values are equal
assert_equalfile() assert that two file contents are equal
assert_notequal() assert that two expressions values are not equal
assert_inrange() assert that an expression is inside a range
assert_match() assert that a pattern matches the value
assert_notmatch() assert that a pattern does not match the value
assert_false() assert that an expression is false
assert_true() assert that an expression is true
assert_exception() assert that a command throws an exception
assert_beeps() assert that a command beeps
assert_nobeep() assert that a command does not cause a beep
assert_fails() assert that a command fails
assert_report() report a test failure

Timers: timer-functions

timer_start() create a timer
timer_pause() pause or unpause a timer
timer_stop() stop a timer

```
        timer_stopall()         stop all timers
        timer_info()            get information about timers
        wait()                  wait for a condition
```

Tags: tag-functions

```
        taglist()               get list of matching tags
        tagfiles()              get a list of tags files
        gettagstack()           get the tag stack of a window
        settagstack()           modify the tag stack of a window
```

Prompt Buffer: promptbuffer-functions

```
        prompt_getprompt()      get the effective prompt text for a buffer
        prompt_setcallback()    set prompt callback for a buffer
        prompt_setinterrupt()   set interrupt callback for a buffer
        prompt_setprompt()      set the prompt text for a buffer
```

Registers: register-functions

```
        getreg()                get contents of a register
        getreginfo()            get information about a register
        getregtype()            get type of a register
        setreg()                set contents and type of a register
        reg_executing()         return the name of the register being executed
        reg_recording()         return the name of the register being recorded
```

Various: various-functions

```
        mode()                  get current editing mode
        visualmode()            last visual mode used
        exists()                check if a variable, function, etc. exists
        has()                   check if a feature is supported in Vim
        changenr()              return number of most recent change
        did_filetype()          check if a FileType autocommand was used
        eventhandler()          check if invoked by an event handler
        getpid()                get process ID of Vim
        getscriptinfo()         get list of sourced Vim scripts
        getstacktrace()         get current stack trace of Vim scripts

        libcall()               call a function in an external library
        libcallnr()             idem, returning a number

        undofile()              get the name of the undo file
        undotree()              return the state of the undo tree for a buffer

        shiftwidth()            effective value of 'shiftwidth'

        wordcount()             get byte/word/char count of buffer

        luaeval()               evaluate Lua expression
        py3eval()               evaluate Python expression
        pyeval()                evaluate Python expression
        pyxeval()               evaluate python_x expression
        rubyeval()              evaluate Ruby expression

        debugbreak()            interrupt a program being debugged
```

41.7 Defining a function

Vim enables you to define your own functions. The basic function declaration
begins as follows:

```
:function {name}({var1}, {var2}, ...)
:  {body}
:endfunction
```

        Note:
        Function names must begin with a capital letter.

Let's define a short function to return the smaller of two numbers. It starts
with this line:

```
:function Min(num1, num2)
```

This tells Vim that the function is named "Min" and it takes two arguments:
"num1" and "num2".
   The first thing you need to do is to check to see which number is smaller:

```
:  if a:num1 < a:num2
```

The special prefix "a:" tells Vim that the variable is a function argument.
Let's assign the variable "smaller" the value of the smallest number:

```
:  if a:num1 < a:num2
:    let smaller = a:num1
:  else
:    let smaller = a:num2
:  endif
```

The variable "smaller" is a local variable. Variables used inside a function
are local unless prefixed by something like "g:", "a:", or "s:".

Note:
To access a global variable from inside a function you must prepend "g:" to it. Thus "g:today" inside a function is used for the global variable "today", and "today" is another variable, local to the function.

You now use the ":return" statement to return the smallest number to the user. Finally, you end the function:

```
:  return smaller
:endfunction
```

The complete function definition is as follows:

```
:function Min(num1, num2)
:  if a:num1 < a:num2
:    let smaller = a:num1
:  else
:    let smaller = a:num2
:  endif
:  return smaller
:endfunction
```

For people who like short functions, this does the same thing:

```
:function Min(num1, num2)
:  if a:num1 < a:num2
:    return a:num1
:  endif
:  return a:num2
:endfunction
```

A user defined function is called in exactly the same way as a built-in function. Only the name is different. The Min function can be used like this:

```
:echo Min(5, 8)
```

Only now will the function be executed and the lines be interpreted by Vim. If there are mistakes, like using an undefined variable or function, you will now get an error message. When defining the function these errors are not detected.

When a function reaches ":endfunction" or ":return" is used without an argument, the function returns zero.

To redefine a function that already exists, use the ! for the ":function" command:

```
:function!  Min(num1, num2, num3)
```

USING A RANGE

The ":call" command can be given a line range. This can have one of two meanings. When a function has been defined with the "range" keyword, it will take care of the line range itself.
The function will be passed the variables "a:firstline" and "a:lastline". These will have the line numbers from the range the function was called with. Example:

```
:function Count_words() range
:  let lnum = a:firstline
:  let n = 0
:  while lnum <= a:lastline
:    let n = n + len(split(getline(lnum)))
:    let lnum = lnum + 1
:  endwhile
:  echo "found " .. n .. " words"
:endfunction
```

You can call this function with:

```
:10,30call Count_words()
```

It will be executed once and echo the number of words.
The other way to use a line range is by defining a function without the "range" keyword. The function will be called once for every line in the range, with the cursor in that line. Example:

```
:function  Number()
:  echo "line " .. line(".") .. " contains: " .. getline(".")
:endfunction
```

If you call this function with:

```
:10,15call Number()
```

The function will be called six times.

VARIABLE NUMBER OF ARGUMENTS

Vim enables you to define functions that have a variable number of arguments.
The following command, for instance, defines a function that must have 1
argument (start) and can have up to 20 additional arguments:

```
:function Show(start, ...)
```

The variable "a:1" contains the first optional argument, "a:2" the second, and
so on. The variable "a:0" contains the number of extra arguments.
   For example:

```
:function Show(start, ...)
:  echohl Title
:  echo "start is " .. a:start
:  echohl None
:  let index = 1
:  while index <= a:0
:    echo "  Arg " .. index .. " is " .. a:{index}
:    let index = index + 1
:  endwhile
:  echo ""
:endfunction
```

This uses the ":echohl" command to specify the highlighting used for the
following ":echo" command. ":echohl None" stops it again. The ":echon"
command works like ":echo", but doesn't output a line break.

You can also use the a:000 variable, it is a List of all the "..." arguments.
See a:000.

LISTING FUNCTIONS

The ":function" command lists the names and arguments of all user-defined
functions:

```
:function
```

```
function Show(start, ...)
function GetVimIndent()
function SetSyn(name)
```

To see what a function does, use its name as an argument for ":function":

```
:function SetSyn
```

```
1     if &syntax == ''
2       let &syntax = a:name
3     endif
endfunction
```

DEBUGGING

The line number is useful for when you get an error message or when debugging.
See debug-scripts about debugging mode.
   You can also set the 'verbose' option to 12 or higher to see all function
calls. Set it to 15 or higher to see every executed line.

DELETING A FUNCTION

To delete the Show() function:

```
:delfunction Show
```

You get an error when the function doesn't exist.

FUNCTION REFERENCES

Sometimes it can be useful to have a variable point to one function or
another. You can do it with the function() function. It turns the name of a
function into a reference:

```
:let result = 0         " or 1
:function! Right()
:  return 'Right!'
:endfunc
:function! Wrong()
:  return 'Wrong!'
:endfunc
:
:if result == 1
:  let Afunc = function('Right')
:else
:  let Afunc = function('Wrong')
:endif
:echo call(Afunc, [])
```

Wrong!

Note that the name of a variable that holds a function reference must start with a capital. Otherwise it could be confused with the name of a builtin function.
The way to invoke a function that a variable refers to is with the call() function. Its first argument is the function reference, the second argument is a List with arguments.

Function references are most useful in combination with a Dictionary, as is explained in the next section.

More information about defining your own functions here: user-function.

41.8 Lists and Dictionaries

So far we have used the basic types String and Number. Vim also supports two composite types: List and Dictionary.

A List is an ordered sequence of things. The things can be any kind of value, thus you can make a List of numbers, a List of Lists and even a List of mixed items. To create a List with three strings:

```
:let alist = ['aap', 'mies', 'noot']
```

The List items are enclosed in square brackets and separated by commas. To create an empty List:

```
:let alist = []
```

You can add items to a List with the add() function:

```
:let alist = []
:call add(alist, 'foo')
:call add(alist, 'bar')
:echo alist
```

['foo', 'bar']

List concatenation is done with +:

```
:echo alist + ['foo', 'bar']
```

['foo', 'bar', 'foo', 'bar']

Or, if you want to extend a List directly:

```
:let alist = ['one']
:call extend(alist, ['two', 'three'])
:echo alist
```

['one', 'two', 'three']

Notice that using add() will have a different effect:

```
:let alist = ['one']
:call add(alist, ['two', 'three'])
:echo alist
```

['one', ['two', 'three']]

The second argument of add() is added as a single item.

FOR LOOP

One of the nice things you can do with a List is iterate over it:

```
:let alist = ['one', 'two', 'three']
:for n in alist
:  echo n
:endfor
```

one
two
three

This will loop over each element in List "alist", assigning the value to
variable "n". The generic form of a for loop is:

```
:for {varname} in {listexpression}
:  {commands}
:endfor
```

To loop a certain number of times you need a List of a specific length. The
range() function creates one for you:

```
:for a in range(3)
:  echo a
:endfor
```

0
1
2

Notice that the first item of the List that range() produces is zero, thus the
last item is one less than the length of the list.
   You can also specify the maximum value, the stride and even go backwards:

```
:for a in range(8, 4, -2)
:  echo a
:endfor
```

8
6
4

A more useful example, looping over lines in the buffer:

```
:for line in getline(1, 20)
:  if line =~ "Date: "
:    echo matchstr(line, 'Date: \zs.*')
:  endif
:endfor
```

This looks into lines 1 to 20 (inclusive) and echoes any date found in there.

DICTIONARIES

A Dictionary stores key-value pairs. You can quickly lookup a value if you
know the key. A Dictionary is created with curly braces:

```
:let uk2nl = {'one': 'een', 'two': 'twee', 'three': 'drie'}
```

Now you can lookup words by putting the key in square brackets:

```
:echo uk2nl['two']
```

twee

The generic form for defining a Dictionary is:

```
{<key> : <value>, ...}
```

An empty Dictionary is one without any keys:

```
{}
```

The possibilities with Dictionaries are numerous. There are various functions
for them as well. For example, you can obtain a list of the keys and loop
over them:

```
:for key in keys(uk2nl)
:  echo key
:endfor
```

three
one
two

You will notice the keys are not ordered. You can sort the list to get a
specific order:

```
:for key in sort(keys(uk2nl))
:  echo key
:endfor
```

one
three
two

But you can never get back the order in which items are defined. For that you
need to use a List, it stores items in an ordered sequence.

DICTIONARY FUNCTIONS

The items in a Dictionary can normally be obtained with an index in square
brackets:

```
:echo uk2nl['one']
```

een

A method that does the same, but without so many punctuation characters:

```
:echo uk2nl.one
```

een

This only works for a key that is made of ASCII letters, digits and the
underscore. You can also assign a new value this way:

```
:let uk2nl.four = 'vier'
:echo uk2nl
```

{'three': 'drie', 'four': 'vier', 'one': 'een', 'two': 'twee'}

And now for something special: you can directly define a function and store a
reference to it in the dictionary:

```
:function uk2nl.translate(line) dict
:  return join(map(split(a:line), 'get(self, v:val, "???")'))
:endfunction
```

Let's first try it out:

```
:echo uk2nl.translate('three two five one')
```

drie twee ??? een

The first special thing you notice is the "dict" at the end of the ":function"
line. This marks the function as being used from a Dictionary. The "self"
local variable will then refer to that Dictionary.
   Now let's break up the complicated return command:

```
split(a:line)
```

The split() function takes a string, chops it into whitespace separated words
and returns a list with these words. Thus in the example it returns:

```
:echo split('three two five one')
```

['three', 'two', 'five', 'one']

This list is the first argument to the map() function. This will go through
the list, evaluating its second argument with "v:val" set to the value of each
item. This is a shortcut to using a for loop. This command:

```
:let alist = map(split(a:line), 'get(self, v:val, "???")')
```

Is equivalent to:

```
:let alist = split(a:line)
:for idx in range(len(alist))
:  let alist[idx] = get(self, alist[idx], "???")
:endfor
```

The get() function checks if a key is present in a Dictionary. If it is, then
the value is retrieved. If it isn't, then the default value is returned, in
the example it's '???'. This is a convenient way to handle situations where a
key may not be present and you don't want an error message.

The join() function does the opposite of split(): it joins together a list of
words, putting a space in between.
  This combination of split(), map() and join() is a nice way to filter a line
of words in a very compact way.

OBJECT ORIENTED PROGRAMMING

Now that you can put both values and functions in a Dictionary, you can
actually use a Dictionary like an object.
   Above we used a Dictionary for translating Dutch to English. We might want
to do the same for other languages. Let's first make an object (aka
Dictionary) that has the translate function, but no words to translate:

```
:let transdict = {}
:function transdict.translate(line) dict
:  return join(map(split(a:line), 'get(self.words, v:val, "???")'))
:endfunction
```

It's slightly different from the function above, using 'self.words' to lookup

word translations. But we don't have a self.words. Thus you could call this an abstract class.

Now we can instantiate a Dutch translation object:

```
:let uk2nl = copy(transdict)
:let uk2nl.words = {'one': 'een', 'two': 'twee', 'three': 'drie'}
:echo uk2nl.translate('three one')
```

drie een

And a German translator:

```
:let uk2de = copy(transdict)
:let uk2de.words = {'one': 'eins', 'two': 'zwei', 'three': 'drei'}
:echo uk2de.translate('three one')
```

drei eins

You see that the copy() function is used to make a copy of the "transdict" Dictionary and then the copy is changed to add the words. The original remains the same, of course.

Now you can go one step further, and use your preferred translator:

```
:if $LANG =~ "de"
:  let trans = uk2de
:else
:  let trans = uk2nl
:endif
:echo trans.translate('one two three')
```

een twee drie

Here "trans" refers to one of the two objects (Dictionaries). No copy is made. More about List and Dictionary identity can be found at list-identity and dict-identity.

Now you might use a language that isn't supported. You can overrule the translate() function to do nothing:

```
:let uk2uk = copy(transdict)
:function! uk2uk.translate(line)
:  return a:line
:endfunction
:echo uk2uk.translate('three one wladiwostok')
```

three one wladiwostok

Notice that a ! was used to overwrite the existing function reference. Now use "uk2uk" when no recognized language is found:

```
:if $LANG =~ "de"
:  let trans = uk2de
:elseif $LANG =~ "nl"
:  let trans = uk2nl
:else
:  let trans = uk2uk
:endif
:echo trans.translate('one two three')
```

one two three

For further reading see Lists and Dictionaries.

41.9 Exceptions

Let's start with an example:

```
:try
:   read ~/templates/pascal.tmpl
:catch /E484:/
:   echo "Sorry, the Pascal template file cannot be found."
:endtry
```

The ":read" command will fail if the file does not exist. Instead of generating an error message, this code catches the error and gives the user a nice message.

For the commands in between ":try" and ":endtry" errors are turned into exceptions. An exception is a string. In the case of an error the string contains the error message. And every error message has a number. In this case, the error we catch contains "E484:". This number is guaranteed to stay the same (the text may change, e.g., it may be translated).

When the ":read" command causes another error, the pattern "E484:" will not match in it. Thus this exception will not be caught and result in the usual error message and execution is aborted.

You might be tempted to do this:

```
:try
:   read ~/templates/pascal.tmpl
:catch
:   echo "Sorry, the Pascal template file cannot be found."
:endtry
```

This means all errors are caught. But then you will not see errors that are useful, such as "E21: Cannot make changes, 'modifiable' is off".

Another useful mechanism is the ":finally" command:

```
:let tmp = tempname()
:try
:   exe ".,$write " .. tmp
:   exe "!filter " .. tmp
:   .,$delete
:   exe "$read " .. tmp
:finally
:   call delete(tmp)
:endtry
```

This filters the lines from the cursor until the end of the file through the "filter" command, which takes a file name argument. No matter if the filtering works, something goes wrong in between ":try" and ":finally" or the user cancels the filtering by pressing CTRL-C, the "call delete(tmp)" is always executed. This makes sure you don't leave the temporary file behind.

More information about exception handling can be found in the reference manual: exception-handling.

41.10 Various remarks

Here is a summary of items that apply to Vim scripts. They are also mentioned elsewhere, but form a nice checklist.

The end-of-line character depends on the system. For Vim scripts it is recommended to always use the Unix fileformat. Lines are then separated with the Newline character. This also works on any other system. That way you can copy your Vim scripts from MS-Windows to Unix and they still work. See :source_crnl. To be sure it is set right, do this before writing the file:

```
:setlocal fileformat=unix
```

When using "dos" fileformat, lines are separated with CR-NL, two characters. The CR character causes various problems, better avoid this.

WHITE SPACE

Blank lines are allowed in a script and ignored.

Leading whitespace characters (blanks and TABs) are ignored, except when using :let-heredoc without "trim".

Trailing whitespace is often ignored, but not always. One command that includes it is map. You have to watch out for that, it can cause hard to understand mistakes. A generic solution is to never use trailing white space, unless you really need it.

To include a whitespace character in the value of an option, it must be escaped by a "\" (backslash) as in the following example:

```
:set tags=my\ nice\ file
```

The same example written as:

```
:set tags=my nice file
```

will issue an error, because it is interpreted as:

```
:set tags=my
:set nice
:set file
```

COMMENTS

The character " (the double quote mark) starts a comment. Everything after and including this character until the end-of-line is considered a comment and is ignored, except for commands that don't consider comments, as shown in examples below. A comment can start on any character position on the line.

There is a little "catch" with comments for some commands. Examples:

```
:abbrev dev development         " shorthand
:map <F3> o#include             " insert include
:execute cmd                    " do it
:!ls *.c                        " list C files
```

The abbreviation "dev" will be expanded to 'development   " shorthand'. The
mapping of <F3> will actually be the whole line after the 'o# ....' including
the '" insert include'. The "execute" command will give an error. The "!"
command will send everything after it to the shell, causing an error for an
unmatched '"' character.
   There can be no comment after ":map", ":abbreviate", ":execute" and "!"
commands (there are a few more commands with this restriction). For the
":map", ":abbreviate" and ":execute" commands there is a trick:

```
:abbrev dev development|" shorthand
:map <F3> o#include|" insert include
:execute cmd                    |" do it
```

With the '|' character the command is separated from the next one. And that
next command is only a comment. For the last command you need to do two
things: :execute and use '|':

```
:exe '!ls *.c'                  |" list C files
```

Notice that there is no white space before the '|' in the abbreviation and
mapping. For these commands, any character until the end-of-line or '|' is
included. As a consequence of this behavior, you don't always see that
trailing whitespace is included:

```
:map <F4> o#include
```

To spot these problems, you can set the 'list' option when editing vimrc
files.

For Unix there is one special way to comment a line, that allows making a Vim
script executable:

```
#!/usr/bin/env vim -S
echo "this is a Vim script"
quit
```

The "#" command by itself lists a line with the line number. Adding an
exclamation mark changes it into doing nothing, so that you can add the shell
command to execute the rest of the file. :#! -S

PITFALLS

Even bigger problem arises in the following example:

```
:map ,ab o#include
:unmap ,ab
```

Here the unmap command will not work, because it tries to unmap ",ab ". This
does not exist as a mapped sequence. An error will be issued, which is very
hard to identify, because the ending whitespace character in ":unmap ,ab " is
not visible.

And this is the same as what happens when one uses a comment after an "unmap"
command:

```
:unmap ,ab     " comment
```

Here the comment part will be ignored. However, Vim will try to unmap
',ab   ', which does not exist. Rewrite it as:

```
:unmap ,ab|    " comment
```

RESTORING THE VIEW

Sometimes you want to make a change and go back to where the cursor was.
Restoring the relative position would also be nice, so that the same line
appears at the top of the window.
   This example yanks the current line, puts it above the first line in the
file and then restores the view:

```
map ,p ma"aYHmbgg"aP`bzt`a
```

What this does:

```
ma"aYHmbgg"aP`bzt`a
```

```
        ma                      set mark a at cursor position
          "aY                   yank current line into register a
             Hmb                go to top line in window and set mark b there
                gg              go to first line in file
                  "aP           put the yanked line above it
```

```
                    `b          go back to top line in display
                      zt        position the text in the window as before
                        `a      go back to saved cursor position
```

PACKAGING

To avoid your function names to interfere with functions that you get from others, use this scheme:

- Prepend a unique string before each function name. I often use an abbreviation. For example, "OW_" is used for the option window functions.
- Put the definition of your functions together in a file. Set a global variable to indicate that the functions have been loaded. When sourcing the file again, first unload the functions.

Example:

```
" This is the XXX package

if exists("XXX_loaded")
  delfun XXX_one
  delfun XXX_two
endif

function XXX_one(a)
        ... body of function ...
endfun

function XXX_two(b)
        ... body of function ...
endfun

let XXX_loaded = 1
```

==========

41.11 Writing a plugin write-plugin

You can write a Vim script in such a way that many people can use it. This is called a plugin. Vim users can drop your script in their plugin directory and use its features right away add-plugin.

There are actually two types of plugins:

```
  global plugins: For all types of files.
filetype plugins: Only for files of a specific type.
```

In this section the first type is explained. Most items are also relevant for writing filetype plugins. The specifics for filetype plugins are in the next section write-filetype-plugin.

NAME

First of all you must choose a name for your plugin. The features provided by the plugin should be clear from its name. And it should be unlikely that someone else writes a plugin with the same name but which does something different. And please limit the name to 8 characters, to avoid problems on old MS-Windows systems.

A script that corrects typing mistakes could be called "typecorr.vim". We will use it here as an example.

For the plugin to work for everybody, it should follow a few guidelines. This will be explained step-by-step. The complete example plugin is at the end.

BODY

Let's start with the body of the plugin, the lines that do the actual work:

```
14      iabbrev teh the
15      iabbrev otehr other
16      iabbrev wnat want
17      iabbrev synchronisation
18              \ synchronization
19      let s:count = 4
```

The actual list should be much longer, of course.

The line numbers have only been added to explain a few things, don't put them in your plugin file!

HEADER

You will probably add new corrections to the plugin and soon have several versions lying around. And when distributing this file, people will want to know who wrote this wonderful plugin and where they can send remarks. Therefore, put a header at the top of your plugin:

```
1       " Vim global plugin for correcting typing mistakes
2       " Last Change:  2000 Oct 15
3       " Maintainer:   Bram Moolenaar <Bram@vim.org>
```

About copyright and licensing: Since plugins are very useful and it's hardly worth restricting their distribution, please consider making your plugin either public domain or use the Vim license. A short note about this near the top of the plugin should be sufficient. Example:

```
4       " License:      This file is placed in the public domain.
```

LINE CONTINUATION, AVOIDING SIDE EFFECTS use-cpo-save

In line 18 above, the line-continuation mechanism is used line-continuation. Users with 'compatible' set will run into trouble here, they will get an error message. We can't just reset 'compatible', because that has a lot of side effects. To avoid this, we will set the 'cpoptions' option to its Vim default value and restore it later. That will allow the use of line-continuation and make the script work for most people. It is done like this:

```
11      let s:save_cpo = &cpo
12      set cpo&vim
..
42      let &cpo = s:save_cpo
43      unlet s:save_cpo
```

We first store the old value of 'cpoptions' in the s:save_cpo variable. At the end of the plugin this value is restored.

Notice that a script-local variable is used s:var. A global variable could already be in use for something else. Always use script-local variables for things that are only used in the script.

NOT LOADING

It's possible that a user doesn't always want to load this plugin. Or the system administrator has dropped it in the system-wide plugin directory, but a user has their own plugin they want to use. Then the user must have a chance to disable loading this specific plugin. This will make it possible:

```
6       if exists("g:loaded_typecorr")
7         finish
8       endif
9       let g:loaded_typecorr = 1
```

This also avoids that when the script is loaded twice it would cause error messages for redefining functions and cause trouble for autocommands that are added twice.

The name is recommended to start with "loaded_" and then the file name of the plugin, literally. The "g:" is prepended just to avoid mistakes when using the variable in a function (without "g:" it would be a variable local to the function).

Using "finish" stops Vim from reading the rest of the file, it's much quicker than using if-endif around the whole file.

MAPPING

Now let's make the plugin more interesting: We will add a mapping that adds a correction for the word under the cursor. We could just pick a key sequence for this mapping, but the user might already use it for something else. To allow the user to define which keys a mapping in a plugin uses, the <Leader> item can be used:

```
22        map <unique> <Leader>a  <Plug>(TypecorrAdd)
```

The "<Plug>(TypecorrAdd)" thing will do the work, more about that further on.

The user can set the "mapleader" variable to the key sequence that they want this mapping to start with. Thus if the user has done:

```
let mapleader = "_"
```

the mapping will define "_a". If the user didn't do this, the default value will be used, which is a backslash. Then a map for "\a" will be defined.

Note that <unique> is used, this will cause an error message if the mapping already happened to exist. :map-<unique>

But what if the user wants to define their own key sequence? We can allow that with this mechanism:

```
21      if !hasmapto('<Plug>(TypecorrAdd)')
22        map <unique> <Leader>a  <Plug>(TypecorrAdd)
23      endif
```

This checks if a mapping to "<Plug>(TypecorrAdd)" already exists, and only defines the mapping from "<Leader>a" if it doesn't. The user then has a chance of putting this in their vimrc file:

```
map ,c  <Plug>(TypecorrAdd)
```

Then the mapped key sequence will be ",c" instead of "_a" or "\a".

PIECES

If a script gets longer, you often want to break up the work in pieces. You can use functions or mappings for this. But you don't want these functions and mappings to interfere with the ones from other scripts. For example, you could define a function Add(), but another script could try to define the same function. To avoid this, we define the function local to the script by prepending it with "s:".

We will define a function that adds a new typing correction:

```
30      function s:Add(from, correct)
31        let to = input("type the correction for " .. a:from .. ": ")
32        exe ":iabbrev " .. a:from .. " " .. to
..
36      endfunction
```

Now we can call the function s:Add() from within this script. If another script also defines s:Add(), it will be local to that script and can only be called from the script it was defined in. There can also be a global Add() function (without the "s:"), which is again another function.

<SID> can be used with mappings. It generates a script ID, which identifies the current script. In our typing correction plugin we use it like this:

```
24      noremap <unique> <script> <Plug>(TypecorrAdd)  <SID>Add
..
28      noremap <SID>Add  :call <SID>Add(expand("<cword>"), 1)<CR>
```

Thus when a user types "\a", this sequence is invoked:

```
\a  ->  <Plug>(TypecorrAdd)  ->  <SID>Add  ->  :call <SID>Add()
```

If another script also maps <SID>Add, it will get another script ID and thus define another mapping.

Note that instead of s:Add() we use <SID>Add() here. That is because the mapping is typed by the user, thus outside of the script. The <SID> is translated to the script ID, so that Vim knows in which script to look for the Add() function.

This is a bit complicated, but it's required for the plugin to work together with other plugins. The basic rule is that you use <SID>Add() in mappings and s:Add() in other places (the script itself, autocommands, user commands).

We can also add a menu entry to do the same as the mapping:

```
26      noremenu <script> Plugin.Add\ Correction      <SID>Add
```

The "Plugin" menu is recommended for adding menu items for plugins. In this case only one item is used. When adding more items, creating a submenu is recommended. For example, "Plugin.CVS" could be used for a plugin that offers CVS operations "Plugin.CVS.checkin", "Plugin.CVS.checkout", etc.

Note that in line 28 ":noremap" is used to avoid that any other mappings cause trouble. Someone may have remapped ":call", for example. In line 24 we also use ":noremap", but we do want "<SID>Add" to be remapped. This is why "<script>" is used here. This only allows mappings which are local to the script. :map-<script> The same is done in line 26 for ":noremenu". :menu-<script>

<SID> AND <Plug> using-<Plug>

Both <SID> and <Plug> are used to avoid that mappings of typed keys interfere with mappings that are only to be used from other mappings. Note the difference between using <SID> and <Plug>:

<Plug> is visible outside of the script. It is used for mappings which the user might want to map a key sequence to. <Plug> is a special code that a typed key will never produce.
To make it very unlikely that other plugins use the same sequence of characters, use this structure: <Plug> scriptname mapname
In our example the scriptname is "Typecorr" and the mapname is "Add".
We wrap the name in parentheses. This results in "<Plug>(TypecorrAdd)". The convention of using PascalCase helps telling scriptname apart from mapname.

Note: the text after "<Plug>" is chosen by the plugin author and is entirely arbitrary. Wrapping the name in parentheses, using PascalCase, or adding a ";" terminator like "<Plug>TypecorrAdd;" are only conventions to keep the name readable and unlikely to clash with other plugins. None of them are required: all that matters is that

the sequence starts with "<Plug>" and is unique.
A mapping whose {lhs} starts with <Plug> is always applied to the
{rhs} of another mapping, even if remapping is explicitly disallowed.
:nore

<SID> is the script ID, a unique identifier for a script.
Internally Vim translates <SID> to "<SNR>123_", where "123" can be any
number. Thus a function "<SID>Add()" will have a name "<SNR>11_Add()"
in one script, and "<SNR>22_Add()" in another. You can see this if
you use the ":function" command to get a list of functions. The
translation of <SID> in mappings is exactly the same, that's how you
can call a script-local function from a mapping.
Note: when <SID> appears in the {rhs} of a mapping, you might consider
using <script>. :map-<script>

USER COMMAND

Now let's add a user command to add a correction:

```
38      if !exists(":Correct")
39        command -nargs=1  Correct  :call s:Add(<q-args>, 0)
40      endif
```

The user command is defined only if no command with the same name already
exists. Otherwise we would get an error here. Overriding the existing user
command with ":command!" is not a good idea, this would probably make the user
wonder why the command they defined themself doesn't work. :command

SCRIPT VARIABLES

When a variable starts with "s:" it is a script variable. It can only be used
inside a script. Outside the script it's not visible. This avoids trouble
with using the same variable name in different scripts. The variables will be
kept as long as Vim is running. And the same variables are used when sourcing
the same script again. s:var

The fun is that these variables can also be used in functions, autocommands
and user commands that are defined in the script. In our example we can add
a few lines to count the number of corrections:

```
19      let s:count = 4
..
30      function s:Add(from, correct)
..
34        let s:count = s:count + 1
35        echo s:count .. " corrections now"
36      endfunction
```

First s:count is initialized to 4 in the script itself. When later the
s:Add() function is called, it increments s:count. It doesn't matter from
where the function was called, since it has been defined in the script, it
will use the local variables from this script.

THE RESULT

Here is the resulting complete example:

```
1      " Vim global plugin for correcting typing mistakes
2      " Last Change:  2000 Oct 15
3      " Maintainer:   Bram Moolenaar <Bram@vim.org>
4      " License:      This file is placed in the public domain.
5
6      if exists("g:loaded_typecorr")
7        finish
8      endif
9      let g:loaded_typecorr = 1
10
11     let s:save_cpo = &cpo
12     set cpo&vim
13
14     iabbrev teh the
15     iabbrev otehr other
16     iabbrev wnat want
17     iabbrev synchronisation
18             \ synchronization
19     let s:count = 4
20
21     if !hasmapto('<Plug>(TypecorrAdd)')
22       map <unique> <Leader>a  <Plug>(TypecorrAdd)
23     endif
24     noremap <unique> <script> <Plug>(TypecorrAdd)  <SID>Add
25
26     noremenu <script> Plugin.Add\ Correction      <SID>Add
27
28     noremap <SID>Add  :call <SID>Add(expand("<cword>"), 1)<CR>
29
30     function s:Add(from, correct)
31       let to = input("type the correction for " .. a:from .. ": ")
32       exe ":iabbrev " .. a:from .. " " .. to
33       if a:correct | exe "normal viws\<C-R>\" \b\e" | endif
34       let s:count = s:count + 1
35       echo s:count .. " corrections now"
36     endfunction
37
38     if !exists(":Correct")
39       command -nargs=1  Correct  :call s:Add(<q-args>, 0)
40     endif
41
42     let &cpo = s:save_cpo
43     unlet s:save_cpo
```

Line 33 wasn't explained yet. It applies the new correction to the word under the cursor. The :normal command is used to use the new abbreviation. Note that mappings and abbreviations are expanded here, even though the function was called from a mapping defined with ":noremap".

Using "unix" for the 'fileformat' option is recommended. The Vim scripts will then work everywhere. Scripts with 'fileformat' set to "dos" do not work on Unix. Also see :source_crnl. To be sure it is set right, do this before writing the file:

```
:set fileformat=unix
```

DOCUMENTATION write-local-help

It's a good idea to also write some documentation for your plugin. Especially when its behavior can be changed by the user. See add-local-help for how they are installed.

Here is a simple example for a plugin help file, called "typecorr.txt":

```
1      *typecorr.txt*  Plugin for correcting typing mistakes
2
3      If you make typing mistakes, this plugin will have them corrected
4      automatically.
5
6      There are currently only a few corrections.  Add your own if you like.
7
8      Mappings:
9      <Leader>a   or   <Plug>(TypecorrAdd)
10             Add a correction for the word under the cursor.
11
12     Commands:
13     :Correct {word}
14             Add a correction for {word}.
15
16                                                  *typecorr-settings*
17     This plugin doesn't have any settings.
```

The first line is actually the only one for which the format matters. It will be extracted from the help file to be put in the "LOCAL ADDITIONS:" section of first line. After adding your help file do ":help" and check that the entries line up nicely.

You can add more tags inside ** in your help file. But be careful not to use existing help tags. You would probably use the name of your plugin in most of them, like "typecorr-settings" in the example.

Using references to other parts of the help in || is recommended. This makes it easy for the user to find associated help.

FILETYPE DETECTION plugin-filetype

If your filetype is not already detected by Vim, you should create a filetype detection snippet in a separate file. It is usually in the form of an autocommand that sets the filetype when the file name matches a pattern. Example:

```
au BufNewFile,BufRead *.foo                    set filetype=foofoo
```

Write this single-line file as "ftdetect/foofoo.vim" in the first directory that appears in 'runtimepath'. For Unix that would be "~/.config/nvim/ftdetect/foofoo.vim". The convention is to use the name of the filetype for the script name.

You can make more complicated checks if you like, for example to inspect the contents of the file to recognize the language. Also see new-filetype.

SUMMARY plugin-special

Summary of special things to use in a plugin:

s:name	Variables local to the script.
<SID>	Script-ID, used for mappings and functions local to the script.
hasmapto()	Function to test if the user already defined a mapping for functionality the script offers.
<Leader>	Value of "mapleader", which the user defines as the keys that plugin mappings start with.
:map <unique>	Give a warning if a mapping already exists.
:noremap <script>	Only remap mappings defined in this script that start with <SID>.
exists(":Cmd")	Check if a user command already exists.

41.12 Writing a filetype plugin write-filetype-plugin ftplugin

A filetype plugin is like a global plugin, except that it sets options and defines mappings for the current buffer only. See add-filetype-plugin for how this type of plugin is used.

First read the section on global plugins above 41.11. All that is said there also applies to filetype plugins. There are a few extras, which are explained here. The essential thing is that a filetype plugin should only have an effect on the current buffer.

DISABLING

If you are writing a filetype plugin to be used by many people, they need a chance to disable loading it. Put this at the top of the plugin:

```
" Only do this when not done yet for this buffer
if exists("b:did_ftplugin")
  finish
endif
let b:did_ftplugin = 1
```

This also needs to be used to avoid that the same plugin is executed twice for the same buffer (happens when using an ":edit" command without arguments).

Now users can disable loading the default plugin completely by making a filetype plugin with only this line:

```
let b:did_ftplugin = 1
```

This does require that the filetype plugin directory comes before $VIMRUNTIME in 'runtimepath'!

If you do want to use the default plugin, but overrule one of the settings, you can write the different setting in a script:

```
setlocal textwidth=70
```

Now write this in the "after" directory, so that it gets sourced after the distributed "vim.vim" ftplugin after-directory. For Unix this would be "~/.config/nvim/after/ftplugin/vim.vim". Note that the default plugin will have set "b:did_ftplugin", but it is ignored here.

OPTIONS

To make sure the filetype plugin only affects the current buffer use the

```
:setlocal
```

command to set options. And only set options which are local to a buffer (see the help for the option to check that). When using :setlocal for global options or options local to a window, the value will change for many buffers, and that is not what a filetype plugin should do.

When an option has a value that is a list of flags or items, consider using "+=" and "-=" to keep the existing value. Be aware that the user may have changed an option value already. First resetting to the default value and then changing it is often a good idea. Example:

```
:setlocal formatoptions& formatoptions+=ro
```

MAPPINGS

To make sure mappings will only work in the current buffer use the

```
:map <buffer>
```

command. This needs to be combined with the two-step mapping explained above. An example of how to define functionality in a filetype plugin:

```
if !hasmapto('<Plug>JavaImport;')
  map <buffer> <unique> <LocalLeader>i <Plug>JavaImport;
endif
noremap <buffer> <unique> <Plug>JavaImport; oimport ""<Left><Esc>
```

hasmapto() is used to check if the user has already defined a map to <Plug>JavaImport;. If not, then the filetype plugin defines the default mapping. This starts with <LocalLeader>, which allows the user to select the key(s) they want filetype plugin mappings to start with. The default is a backslash.
"<unique>" is used to give an error message if the mapping already exists or overlaps with an existing mapping.
:noremap is used to avoid that any other mappings that the user has defined interferes. You might want to use ":noremap <script>" to allow remapping mappings defined in this script that start with <SID>.

The user must have a chance to disable the mappings in a filetype plugin, without disabling everything. Here is an example of how this is done for a plugin for the mail filetype:

```
" Add mappings, unless the user didn't want this.
if !exists("no_plugin_maps") && !exists("no_mail_maps")
  " Quote text by inserting "> "
  if !hasmapto('<Plug>MailQuote;')
    vmap <buffer> <LocalLeader>q <Plug>MailQuote;
    nmap <buffer> <LocalLeader>q <Plug>MailQuote;
  endif
  vnoremap <buffer> <Plug>MailQuote; :s/^/> /<CR>
  nnoremap <buffer> <Plug>MailQuote; :.,$s/^/> /<CR>
endif
```

Two global variables are used:
no_plugin_maps   disables mappings for all filetype plugins
no_mail_maps   disables mappings for the "mail" filetype

USER COMMANDS

To add a user command for a specific file type, so that it can only be used in one buffer, use the "-buffer" argument to :command. Example:

```
:command -buffer  Make  make %:r.s
```

VARIABLES

A filetype plugin will be sourced for each buffer of the type it's for. Local script variables s:var will be shared between all invocations. Use local buffer variables b:var if you want a variable specifically for one buffer.

FUNCTIONS

When defining a function, this only needs to be done once. But the filetype plugin will be sourced every time a file with this filetype will be opened. This construct makes sure the function is only defined once:

```
:if !exists("*s:Func")
:  function s:Func(arg)
:    ...
:  endfunction
:endif
```

UNDO undo_indent undo_ftplugin

When the user does ":setfiletype xyz" the effect of the previous filetype should be undone. Set the b:undo_ftplugin variable to the commands that will undo the settings in your filetype plugin. Example:

```
let b:undo_ftplugin = "setlocal fo< com< tw< commentstring<"
        \ .. "| unlet b:match_ignorecase b:match_words b:match_skip"
```

Using ":setlocal" with "<" after the option name resets the option to its global value. That is mostly the best way to reset the option value.

This does require removing the "C" flag from 'cpoptions' to allow line continuation, as mentioned above use-cpo-save.

For undoing the effect of an indent script, the b:undo_indent variable should be set accordingly.

RECOMMENDED STYLE ft_recommended_style

A filetype plugin or indent script may set options that reflect a recommended or commonly used style for that filetype, rather than a strict requirement of the language. Since not every user wants these stylistic settings, guard them so they can be disabled.

Check two variables, the filetype-specific one first, then the general one, defaulting to enabled (1):

```
if get(g:, '<filetype>_recommended_style',
        \ get(g:, 'filetype_recommended_style', 1))
  " set the recommended style options here
endif
```

Replace <filetype> with the actual filetype name, e.g. "python" gives g:python_recommended_style.

This lets the user turn recommended style settings off for all filetypes:

```
let g:filetype_recommended_style = 0
```

or for one filetype while leaving the rest enabled:

```
let g:python_recommended_style = 0
```

The filetype-specific variable takes precedence over the general one, so a user can disable styling everywhere with g:filetype_recommended_style and re-enable it for a single filetype by setting g:<filetype>_recommended_style to 1 (or vice versa).

Settings that are required for the language to work correctly (not merely stylistic) should not be placed behind this guard.

FILE NAME

The filetype must be included in the file name ftplugin-name. Use one of these three forms:

```
.../ftplugin/stuff.vim
.../ftplugin/stuff_foo.vim
.../ftplugin/stuff/bar.vim
```

"stuff" is the filetype, "foo" and "bar" are arbitrary names.

SUMMARY ftplugin-special

Summary of special things to use in a filetype plugin:

<LocalLeader>	Value of "maplocalleader", which the user defines as the keys that filetype plugin mappings start with.
:map <buffer>	Define a mapping local to the buffer.
:noremap <script>	Only remap mappings defined in this script that start with <SID>.
:setlocal	Set an option for the current buffer only.
:command -buffer	Define a user command local to the buffer.
exists("*s:Func")	Check if a function was already defined.

Also see plugin-special, the special things used for all plugins.

41.13 Writing a compiler plugin write-compiler-plugin

A compiler plugin sets options for use with a specific compiler. The user can load it with the :compiler command. The main use is to set the 'errorformat' and 'makeprg' options.

Easiest is to have a look at examples. This command will edit all the default compiler plugins:

```
:next $VIMRUNTIME/compiler/*.vim
```

Use :next to go to the next plugin file.

There are two special items about these files. First is a mechanism to allow a user to overrule or add to the default file. The default files start with:

```
:if exists("current_compiler")
:  finish
:endif
:let current_compiler = "mine"
```

When you write a compiler file and put it in your personal runtime directory (e.g., ~/.config/nvim/compiler for Unix), you set the "current_compiler" variable to make the default file skip the settings.

:CompilerSet

The second mechanism is to use ":set" for ":compiler!" and ":setlocal" for ":compiler". Vim defines the ":CompilerSet" user command for this. This is an example:

```
CompilerSet errorformat&               " use the default 'errorformat'
CompilerSet makeprg=nmake
```

When you write a compiler plugin for the Vim distribution or for a system-wide runtime directory, use the mechanism mentioned above. When "current_compiler" was already set by a user plugin nothing will be done.

When you write a compiler plugin to overrule settings from a default plugin, don't check "current_compiler". This plugin is supposed to be loaded last, thus it should be in a directory at the end of 'runtimepath'. For Unix that could be ~/.config/nvim/after/compiler.

41.14 Writing a plugin that loads quickly write-plugin-quickload

A plugin may grow and become quite long. The startup delay may become noticeable, while you hardly ever use the plugin. Then it's time for a quickload plugin.

The basic idea is that the plugin is loaded twice. The first time user commands and mappings are defined that offer the functionality. The second time the functions that implement the functionality are defined.

It may sound surprising that quickload means loading a script twice. What we mean is that it loads quickly the first time, postponing the bulk of the script to the second time, which only happens when you actually use it. When you always use the functionality it actually gets slower!

Note that since Vim 7 there is an alternative: use the autoload functionality 41.15.

The following example shows how it's done:

```
" Vim global plugin for demonstrating quick loading
" Last Change:  2005 Feb 25
" Maintainer:   Bram Moolenaar <Bram@vim.org>
" License:      This file is placed in the public domain.

if !exists("s:did_load")
        command -nargs=* BNRead  call BufNetRead(<f-args>)
        map <F19> :call BufNetWrite('something')<CR>

        let s:did_load = 1
        exe 'au FuncUndefined BufNet* source ' .. expand('<script>')
        finish
endif

function BufNetRead(...)
        echo 'BufNetRead(' .. string(a:000) .. ')'
        " read functionality here
endfunction

function BufNetWrite(...)
        echo 'BufNetWrite(' .. string(a:000) .. ')'
        " write functionality here
endfunction
```

When the script is first loaded "s:did_load" is not set. The commands between the "if" and "endif" will be executed. This ends in a :finish command, thus the rest of the script is not executed.

The second time the script is loaded "s:did_load" exists and the commands after the "endif" are executed. This defines the (possible long) BufNetRead() and BufNetWrite() functions.

If you drop this script in your plugin directory Vim will execute it on startup. This is the sequence of events that happens:

1. The "BNRead" command is defined and the <F19> key is mapped when the script is sourced at startup. A FuncUndefined autocommand is defined. The ":finish" command causes the script to terminate early.

2. The user types the BNRead command or presses the <F19> key. The
   BufNetRead() or BufNetWrite() function will be called.
3. Vim can't find the function and triggers the FuncUndefined autocommand
   event. Since the pattern "BufNet*" matches the invoked function, the
   command "source fname" will be executed. "fname" will be equal to the name
   of the script, no matter where it is located, because it comes from
   expanding "<script>" (see expand()).
4. The script is sourced again, the "s:did_load" variable exists and the
   functions are defined.

Notice that the functions that are loaded afterwards match the pattern in the
FuncUndefined autocommand. You must make sure that no other plugin defines
functions that match this pattern.

41.15 Writing library scripts write-library-script

Some functionality will be required in several places. When this becomes more
than a few lines you will want to put it in one script and use it from many
scripts. We will call that one script a library script.

Manually loading a library script is possible, so long as you avoid loading it
when it's already done. You can do this with the exists() function.
Example:

```
if !exists('*MyLibFunction')
   runtime library/mylibscript.vim
endif
call MyLibFunction(arg)
```

Here you need to know that MyLibFunction() is defined in a script
"library/mylibscript.vim" in one of the directories in 'runtimepath'.

To make this a bit simpler Vim offers the autoload mechanism. Then the
example looks like this:

```
call mylib#myfunction(arg)
```

That's a lot simpler, isn't it? Vim will recognize the function name and when
it's not defined search for the script "autoload/mylib.vim" in 'runtimepath'.
That script must define the "mylib#myfunction()" function.

You can put many other functions in the mylib.vim script, you are free to
organize your functions in library scripts. But you must use function names
where the part before the '#' matches the script name. Otherwise Vim would
not know what script to load.

If you get really enthusiastic and write lots of library scripts, you may
want to use subdirectories. Example:

```
call netlib#ftp#read('somefile')
```

For Unix the library script used for this could be:

```
~/.config/nvim/autoload/netlib/ftp.vim
```

Where the function is defined like this:

```
function netlib#ftp#read(fname)
        "  Read the file fname through ftp
endfunction
```

Notice that the name the function is defined with is exactly the same as the
name used for calling the function. And the part before the last '#'
exactly matches the subdirectory and script name.

You can use the same mechanism for variables:

```
let weekdays = dutch#weekdays
```

This will load the script "autoload/dutch.vim", which should contain something
like:

```
let dutch#weekdays = ['zondag', 'maandag', 'dinsdag', 'woensdag',
        \ 'donderdag', 'vrijdag', 'zaterdag']
```

Further reading: autoload.

41.16 Distributing Vim scripts distribute-script

Vim users will look for scripts on the Vim website: https://www.vim.org.
If you made something that is useful for others, share it!

Vim scripts can be used on any system. There might not be a tar or gzip
command. If you want to pack files together and/or compress them the "zip"
utility is recommended.

Next chapter: usr_42.txt Add new menus

Copyright: see manual-copyright

Add new menus

By now you know that Vim is very flexible. This includes the menus used in the GUI. You can define your own menu entries to make certain commands easily accessible. This is for mouse-happy users only.

42.1 Introduction
42.2 Menu commands
42.3 Various
42.4 Toolbar and popup menus

Next chapter: usr_43.txt Using filetypes
Previous chapter: usr_41.txt Write a Vim script
Table of contents: usr_toc.txt

Introduction

The menus that Vim uses are defined in the file "$VIMRUNTIME/menu.vim". If you want to write your own menus, you might first want to look through that file.

To define a menu item, use the ":menu" command. The basic form of this command is as follows:

```
:menu {menu-item} {keys}
```

The {menu-item} describes where on the menu to put the item. A typical {menu-item} is "File.Save", which represents the item "Save" under the "File" menu. A dot is used to separate the names. Example:

```
:menu File.Save  :update<CR>
```

The ":update" command writes the file when it was modified.

You can add another level: "Edit.Settings.Shiftwidth" defines a submenu "Settings" under the "Edit" menu, with an item "Shiftwidth". You could use even deeper levels. Don't use this too much, you need to move the mouse quite a bit to use such an item.

The ":menu" command is very similar to the ":map" command: the left side specifies how the item is triggered and the right hand side defines the characters that are executed. {keys} are characters, they are used just like you would have typed them. Thus in Insert mode, when {keys} is plain text, that text is inserted.

ACCELERATORS

The ampersand character (&) is used to indicate an accelerator. For instance, you can use Alt-F to select "File" and S to select "Save". (The 'winaltkeys' option may disable this though!). Therefore, the {menu-item} looks like "&File.&Save". The accelerator characters will be underlined in the menu.

You must take care that each key is used only once in each menu. Otherwise you will not know which of the two will actually be used. Vim doesn't warn you for this.

PRIORITIES

The actual definition of the File.Save menu item is as follows:

```
:menu 10.340 &File.&Save<Tab>:w  :confirm w<CR>
```

The number 10.340 is called the priority number. It is used by the editor to decide where it places the menu item. The first number (10) indicates the position on the menu bar. Lower numbered menus are positioned to the left, higher numbers to the right.

These are the priorities used for the standard menus:

```
         10    20     40     50      60       70               9999

       +-------------------------------------------------------------+
       | File  Edit  Tools  Syntax  Buffers  Window            Help |
       +-------------------------------------------------------------+
```

Notice that the Help menu is given a very high number, to make it appear on the far right.

The second number (340) determines the location of the item within the pull-down menu. Lower numbers go on top, higher number on the bottom. These are the priorities in the File menu:

```
           +-----------------+
10.310  |Open...          |
10.320  |Split-Open...    |
10.325  |New              |
10.330  |Close            |
10.335  |---------------- |
10.340  |Save             |
10.350  |Save As...       |
10.400  |---------------- |
10.410  |Split Diff with  |
10.420  |Split Patched By |
10.500  |---------------- |
10.510  |Print            |
10.600  |---------------- |
10.610  |Save-Exit        |
10.620  |Exit             |
           +-----------------+
```

Notice that there is room in between the numbers. This is where you can insert your own items, if you really want to (it's often better to leave the standard menus alone and add a new menu for your own items).
   When you create a submenu, you can add another ".number" to the priority. Thus each name in {menu-item} has its priority number.

SPECIAL CHARACTERS

The {menu-item} in this example is "&File.&Save<Tab>:w". This brings up an important point: {menu-item} must be one word. If you want to put a dot, space or tabs in the name, you either use the <> notation (<Space> and <Tab>, for instance) or use the backslash (\) escape.

```
:menu 10.305 &File.&Do\ It\.\.\. :exit<CR>
```

In this example, the name of the menu item "Do It..." contains a space and the command is ":exit<CR>".

The <Tab> character in a menu name is used to separate the part that defines the menu name from the part that gives a hint to the user. The part after the <Tab> is displayed right aligned in the menu. In the File.Save menu the name used is "&File.&Save<Tab>:w". Thus the menu name is "File.Save" and the hint is ":w".

SEPARATORS

The separator lines, used to group related menu items together, can be defined by using a name that starts and ends in a '-'. For example "-sep-". When using several separators the names must be different. Otherwise the names don't matter.
   The command from a separator will never be executed, but you have to define one anyway. A single colon will do. Example:

```
:amenu 20.510 Edit.-sep3- :
```

42.2 Menu commands

You can define menu items that exist for only certain modes. This works just like the variations on the ":map" command:

```
        :menu           Normal, Visual and Operator-pending mode
        :nmenu          Normal mode
        :vmenu          Visual mode
        :omenu          Operator-pending mode
        :menu!          Insert and Command-line mode
        :imenu          Insert mode
        :cmenu          Command-line mode
        :tlmenu         Terminal mode
        :amenu          All modes (except for Terminal mode)
```

To avoid that the commands of a menu item are being mapped, use the command ":noremenu", ":nnoremenu", ":anoremenu", etc.

USING :AMENU

The ":amenu" command is a bit different. It assumes that the {keys} you give are to be executed in Normal mode. When Vim is in Visual or Insert mode when the menu is used, Vim first has to go back to Normal mode. ":amenu" inserts a CTRL-C or CTRL-O for you. For example, if you use this command:

```
:amenu  90.100 Mine.Find\ Word  *
```

Then the resulting menu commands will be:

```
        Normal mode:            *
        Visual mode:            CTRL-C *
        Operator-pending mode:  CTRL-C *
        Insert mode:            CTRL-O *
        Command-line mode:      CTRL-C *
```

When in Command-line mode the CTRL-C will abandon the command typed so far.
In Visual and Operator-pending mode CTRL-C will stop the mode. The CTRL-O in
Insert mode will execute the command and then return to Insert mode.
   CTRL-O only works for one command. If you need to use two or more
commands, put them in a function and call that function. Example:

```
:amenu  Mine.Next\ File  :call <SID>NextFile()<CR>
:function <SID>NextFile()
:  next
:  1/^Code
:endfunction
```

This menu entry goes to the next file in the argument list with ":next". Then
it searches for the line that starts with "Code".
   The <SID> before the function name is the script ID. This makes the
function local to the current Vim script file. This avoids problems when a
function with the same name is defined in another script file. See <SID>.

SILENT MENUS

The menu executes the {keys} as if you typed them. For a ":" command this
means you will see the command being echoed on the command line. If it's a
long command, the hit-Enter prompt will appear. That can be very annoying!
   To avoid this, make the menu silent. This is done with the <silent>
argument. For example, take the call to NextFile() in the previous example.
When you use this menu, you will see this on the command line:

:call <SNR>34_NextFile()

To avoid this text on the command line, insert "<silent>" as the first
argument:

```
:amenu <silent> Mine.Next\ File :call <SID>NextFile()<CR>
```

Don't use "<silent>" too often. It is not needed for short commands. If you
make a menu for someone else, being able to see the executed command will
give them a hint about what they could have typed, instead of using the mouse.

LISTING MENUS

When a menu command is used without a {keys} part, it lists the already
defined menus. You can specify a {menu-item}, or part of it, to list specific
menus. Example:

```
:amenu
```

This lists all menus. That's a long list! Better specify the name of a menu
to get a shorter list:

```
:amenu Edit
```

This lists only the "Edit" menu items for all modes. To list only one
specific menu item for Insert mode:

```
:imenu Edit.Undo
```

Take care that you type exactly the right name. Case matters here. But the
'&' for accelerators can be omitted. The <Tab> and what comes after it can be
left out as well.

DELETING MENUS

To delete a menu, the same command is used as for listing, but with "menu"
changed to "unmenu". Thus ":menu" becomes, ":unmenu", ":nmenu" becomes
":nunmenu", etc. To delete the "Tools.Make" item for Insert mode:

```
:iunmenu Tools.Make
```

You can delete a whole menu, with all its items, by using the menu name.
Example:

```
:aunmenu Syntax
```

This deletes the Syntax menu and all the items in it.

42.3 Various

You can change the appearance of the menus with flags in 'guioptions'. In the
default value they are all included, except "M". You can remove a flag with a
command like:

```
:set guioptions-=m
```

```
       m              When removed the menubar is not displayed.
```

M When added the default menus are not loaded.

g When removed the inactive menu items are not made grey but are completely removed. (Does not work on all systems.)

For translating menu items, see :menutrans.

Since the mouse has to be used to select a menu item, it is a good idea to use the ":browse" command for selecting a file. And ":confirm" to get a dialog instead of an error message, e.g., when the current buffer contains changes. These two can be combined:

```
:amenu File.Open  :browse confirm edit<CR>
```

The ":browse" makes a file browser appear to select the file to edit. The ":confirm" will pop up a dialog when the current buffer has changes. You can then select to save the changes, throw them away or cancel the command.

For more complicated items, the confirm() and inputdialog() functions can be used. The default menus contain a few examples.

42.4 Toolbar and popup menus

There are two special menus: ToolBar and PopUp. Items that start with these names do not appear in the normal menu bar.

TOOLBAR

The toolbar appears only when the "T" flag is included in the 'guioptions' option.

The toolbar uses icons rather than text to represent the command. For example, the {menu-item} named "ToolBar.New" causes the "New" icon to appear on the toolbar.

The Vim editor has 28 built-in icons. You can find a table here: builtin-tools. Most of them are used in the default toolbar. You can redefine what these items do (after the default menus are setup).

You can add another bitmap for a toolbar item. Or define a new toolbar item with a bitmap. For example, define a new toolbar item with:

```
:tmenu ToolBar.Compile  Compile the current file
:amenu ToolBar.Compile  :!cc %:S -o %:r:S<CR>
```

Now you need to create the icon. For MS-Windows it must be in bitmap format, with the name "Compile.bmp". For Unix XPM format is used, the file name is "Compile.xpm". The size must be 18 by 18 pixels. On MS-Windows other sizes can be used as well, but it will look ugly.

Put the bitmap in the directory "bitmaps" in one of the directories from 'runtimepath'. E.g., for Unix "~/.config/nvim/bitmaps/Compile.xpm".

You can define tooltips for the items in the toolbar. A tooltip is a short text that explains what a toolbar item will do. For example "Open file". It appears when the mouse pointer is on the item, without moving for a moment. This is very useful if the meaning of the picture isn't that obvious. Example:

```
:tmenu ToolBar.Make  Run make in the current directory
```

Note:
Pay attention to the case used. "Toolbar" and "toolbar" are different from "ToolBar"!

To remove a tooltip, use the :tunmenu command.

The 'toolbar' option can be used to display text instead of a bitmap, or both text and a bitmap. Most people use just the bitmap, since the text takes quite a bit of space.

POPUP MENU

The popup menu pops up where the mouse pointer is. On MS-Windows you activate it by clicking the right mouse button. Then you can select an item with the left mouse button. On Unix the popup menu is used by pressing and holding the right mouse button.

The popup menu only appears when the 'mousemodel' has been set to "popup" or "popup_setpos". The difference between the two is that "popup_setpos" moves the cursor to the mouse pointer position. When clicking inside a selection, the selection will be used unmodified. When there is a selection but you click outside of it, the selection is removed.

There is a separate popup menu for each mode. Thus there are never grey items like in the normal menus.

What is the meaning of life, the universe and everything? 42
Douglas Adams, the only person who knew what this question really was about is now dead, unfortunately. So now you might wonder what the meaning of death is...

Next chapter: usr_43.txt Using filetypes

Copyright: see manual-copyright

Using filetypes

When you are editing a file of a certain type, for example a C program or a shell script, you often use the same option settings and mappings. You quickly get tired of manually setting these each time. This chapter explains how to do it automatically.

43.1 Plugins for a filetype
43.2 Adding a filetype

Next chapter: usr_44.txt Your own syntax highlighted
Previous chapter: usr_42.txt Add new menus
Table of contents: usr_toc.txt

Plugins for a filetype filetype-plugin

How to start using filetype plugins has already been discussed here: add-filetype-plugin. But you probably are not satisfied with the default settings, because they have been kept minimal. Suppose that for C files you want to set the 'softtabstop' option to 4 and define a mapping to insert a three-line comment. You do this with only two steps:

your-runtime-dir

1. Create your own runtime directory. On Unix this usually is "~/.config/nvim". In this directory create the "ftplugin" directory:

```
mkdir -p ~/.config/nvim/ftplugin
```

   When you are not on Unix, check the value of the 'runtimepath' option to see where Vim will look for the "ftplugin" directory:

```
set runtimepath?
```

   You would normally use the first directory name (before the first comma). You might want to prepend a directory name to the 'runtimepath' option in your init.vim file if you don't like the default value.
2. Create the file "~/.config/nvim/ftplugin/c.vim", with the contents:

```
setlocal softtabstop=4
noremap <buffer> <LocalLeader>c o/**************<CR><CR>/<Esc>
let b:undo_ftplugin = "setl softtabstop< | unmap <buffer> <LocalLeader>c"
```

Try editing a C file. You should notice that the 'softtabstop' option is set to 4. But when you edit another file it's reset to the default zero. That is because the ":setlocal" command was used. This sets the 'softtabstop' option only locally to the buffer. As soon as you edit another buffer, it will be set to the value set for that buffer. For a new buffer it will get the default value or the value from the last ":set" command.

Likewise, the mapping for "\c" will disappear when editing another buffer. The ":map <buffer>" command creates a mapping that is local to the current buffer. This works with any mapping command: ":map!", ":vmap", etc. The <LocalLeader> in the mapping is replaced with the value of the "maplocalleader" variable.

The line to set b:undo_ftplugin is for when the filetype is set to another value. In that case you will want to undo your preferences. The b:undo_ftplugin variable is executed as a command. Watch out for characters with a special meaning inside a string, such as a backslash.

You can find examples for filetype plugins in this directory:

```
$VIMRUNTIME/ftplugin/
```

More details about writing a filetype plugin can be found here: write-plugin.

43.2 Adding a filetype

If you are using a type of file that is not recognized by Vim, this is how to get it recognized. You need a runtime directory of your own. See your-runtime-dir above.

Create a file "filetype.vim" which contains an autocommand for your filetype. (Autocommands were explained in section 40.3.) Example:

```
augroup filetypedetect
au BufNewFile,BufRead *.xyz     setf xyz
augroup END
```

This will recognize all files that end in ".xyz" as the "xyz" filetype. The ":augroup" commands put this autocommand in the "filetypedetect" group. This allows removing all autocommands for filetype detection when doing ":filetype off". The "setf" command will set the 'filetype' option to its argument,

unless it was set already. This will make sure that 'filetype' isn't set twice.

You can use many different patterns to match the name of your file. Directory names can also be included. See autocmd-pattern. For example, the files under "/usr/share/scripts/" are all "ruby" files, but don't have the expected file name extension. Adding this to the example above:

```
augroup filetypedetect
au BufNewFile,BufRead *.xyz                     setf xyz
au BufNewFile,BufRead /usr/share/scripts/*      setf ruby
augroup END
```

However, if you now edit a file /usr/share/scripts/README.txt, this is not a ruby file. The danger of a pattern ending in "*" is that it quickly matches too many files. To avoid trouble with this, put the filetype.vim file in another directory, one that is at the end of 'runtimepath'. For Unix for example, you could use "~/.config/nvim/after/filetype.vim".
   You now put the detection of text files in ~/.config/nvim/filetype.vim:

```
augroup filetypedetect
au BufNewFile,BufRead *.txt                     setf text
augroup END
```

That file is found in 'runtimepath' first. Then use this in ~/.config/nvim/after/filetype.vim, which is found last:

```
augroup filetypedetect
au BufNewFile,BufRead /usr/share/scripts/*      setf ruby
augroup END
```

What will happen now is that Vim searches for "filetype.vim" files in each directory in 'runtimepath'. First ~/.config/nvim/filetype.vim is found. The autocommand to catch *.txt files is defined there. Then Vim finds the filetype.vim file in $VIMRUNTIME, which is halfway 'runtimepath'. Finally ~/.config/nvim/after/filetype.vim is found and the autocommand for detecting ruby files in /usr/share/scripts is added.
   When you now edit /usr/share/scripts/README.txt, the autocommands are checked in the order in which they were defined. The *.txt pattern matches, thus "setf text" is executed to set the filetype to "text". The pattern for ruby matches too, and the "setf ruby" is executed. But since 'filetype' was already set to "text", nothing happens here.
   When you edit the file /usr/share/scripts/foobar the same autocommands are checked. Only the one for ruby matches and "setf ruby" sets 'filetype' to ruby.

RECOGNIZING BY CONTENTS

If your file cannot be recognized by its file name, you might be able to recognize it by its contents. For example, many script files start with a line like:

#!/bin/xyz

To recognize this script create a file "scripts.vim" in your runtime directory (same place where filetype.vim goes). It might look like this:

```
if did_filetype()
  finish
endif
if getline(1) =~ '^#!.*[/\\]xyz\>'
  setf xyz
endif
```

The first check with did_filetype() is to avoid that you will check the contents of files for which the filetype was already detected by the file name. That avoids wasting time on checking the file when the "setf" command won't do anything.
   The scripts.vim file is sourced by an autocommand in the default filetype.vim file. Therefore, the order of checks is:

1. filetype.vim files before $VIMRUNTIME in 'runtimepath'
2. first part of $VIMRUNTIME/filetype.vim
3. all scripts.vim files in 'runtimepath'
4. remainder of $VIMRUNTIME/filetype.vim
5. filetype.vim files after $VIMRUNTIME in 'runtimepath'

If this is not sufficient for you, add an autocommand that matches all files and sources a script or executes a function to check the contents of the file.

Next chapter: usr_44.txt Your own syntax highlighted
Copyright: see manual-copyright

VIM USER MANUAL by Bram Moolenaar

Your own syntax highlighted

Vim comes with highlighting for a couple of hundred different file types. If the file you are editing isn't included, read this chapter to find out how to get this type of file highlighted. Also see :syn-define in the reference manual.

44.1 Basic syntax commands
44.2 Keywords
44.3 Matches
44.4 Regions
44.5 Nested items
44.6 Following groups
44.7 Other arguments
44.8 Clusters
44.9 Including another syntax file
44.10 Synchronizing
44.11 Installing a syntax file
44.12 Portable syntax file layout

Next chapter: usr_45.txt Select your language
Previous chapter: usr_43.txt Using filetypes
Table of contents: usr_toc.txt

Basic syntax commands

Using an existing syntax file to start with will save you a lot of time. Try finding a syntax file in $VIMRUNTIME/syntax for a language that is similar. These files will also show you the normal layout of a syntax file. To understand it, you need to read the following.

Let's start with the basic arguments. Before we start defining any new syntax, we need to clear out any old definitions:

```
:syntax clear
```

This isn't required in the final syntax file, but very useful when experimenting.

There are more simplifications in this chapter. If you are writing a syntax file to be used by others, read all the way through the end to find out the details.

LISTING DEFINED ITEMS

To check which syntax items are currently defined, use this command:

```
:syntax
```

You can use this to check which items have actually been defined. Quite useful when you are experimenting with a new syntax file. It also shows the colors used for each item, which helps to find out what is what.

To list the items in a specific syntax group use:

```
:syntax list {group-name}
```

This also can be used to list clusters (explained in 44.8). Just include the @ in the name.

MATCHING CASE

Some languages are not case sensitive, such as Pascal. Others, such as C, are case sensitive. You need to tell which type you have with the following commands:

```
:syntax case match
:syntax case ignore
```

The "match" argument means that Vim will match the case of syntax elements. Therefore, "int" differs from "Int" and "INT". If the "ignore" argument is used, the following are equivalent: "Procedure", "PROCEDURE" and "procedure".

The ":syntax case" commands can appear anywhere in a syntax file and affect the syntax definitions that follow. In most cases, you have only one ":syntax case" command in your syntax file; if you work with an unusual language that contains both case-sensitive and non-case-sensitive elements, however, you can scatter the ":syntax case" command throughout the file.

44.2 Keywords

The most basic syntax elements are keywords. To define a keyword, use the following form:

```
:syntax keyword {group} {keyword} ...
```

The {group} is the name of a syntax group. With the ":highlight" command you

can assign colors to a {group}. The {keyword} argument is an actual keyword.
Here are a few examples:

```
:syntax keyword xType int long char
:syntax keyword xStatement if then else endif
```

This example uses the group names "xType" and "xStatement". By convention,
each group name is prefixed by the filetype for the language being defined.
This example defines syntax for the x language (eXample language without an
interesting name). In a syntax file for "csh" scripts the name "cshType"
would be used. Thus the prefix is equal to the value of 'filetype'.
   These commands cause the words "int", "long" and "char" to be highlighted
one way and the words "if", "then", "else" and "endif" to be highlighted
another way. Now you need to connect the x group names to standard Vim
names. You do this with the following commands:

```
:highlight link xType Type
:highlight link xStatement Statement
```

This tells Vim to highlight "xType" like "Type" and "xStatement" like
"Statement". See group-name for the standard names.

UNUSUAL KEYWORDS

The characters used in a keyword must be in the 'iskeyword' option. If you
use another character, the word will never match. Vim doesn't give a warning
message for this.
   The x language uses the '-' character in keywords. This is how it's done:

```
:setlocal iskeyword+=-
:syntax keyword xStatement when-not
```

The ":setlocal" command is used to change 'iskeyword' only for the current
buffer. Still it does change the behavior of commands like "w" and "*". If
that is not wanted, don't define a keyword but use a match (explained in the
next section).

The x language allows for abbreviations. For example, "next" can be
abbreviated to "n", "ne" or "nex". You can define them by using this command:

```
:syntax keyword xStatement n[ext]
```

This doesn't match "nextone", keywords always match whole words only.

44.3 Matches

Consider defining something a bit more complex. You want to match ordinary
identifiers. To do this, you define a match syntax item. This one matches
any word consisting of only lowercase letters:

```
:syntax match xIdentifier /\<\l\+\>/
```

> **Note:**
> Keywords overrule any other syntax item. Thus the keywords "if",
> "then", etc., will be keywords, as defined with the ":syntax keyword"
> commands above, even though they also match the pattern for
> xIdentifier.

The part at the end is a pattern, like it's used for searching. The // is
used to surround the pattern (like how it's done in a ":substitute" command).
You can use any other character, like a plus or a quote.

Now define a match for a comment. In the x language it is anything from # to
the end of a line:

```
:syntax match xComment /#.*/
```

Since you can use any search pattern, you can highlight very complex things
with a match item. See pattern for help on search patterns.

44.4 Regions

In the example x language, strings are enclosed in double quotation marks (").
To highlight strings you define a region. You need a region start (double
quote) and a region end (double quote). The definition is as follows:

```
:syntax region xString start=/"/ end=/"/
```

The "start" and "end" directives define the patterns used to find the start
and end of the region. But what about strings that look like this?

"A string with a double quote (\") in it"

This creates a problem: The double quotation marks in the middle of the string
will end the region. You need to tell Vim to skip over any escaped double
quotes in the string. Do this with the skip keyword:

```
:syntax region xString start=/"/ skip=/\\"/ end=/"/
```

The double backslash matches a single backslash, since the backslash is a special character in search patterns.

When to use a region instead of a match? The main difference is that a match item is a single pattern, which must match as a whole. A region starts as soon as the "start" pattern matches. Whether the "end" pattern is found or not doesn't matter. Thus when the item depends on the "end" pattern to match, you cannot use a region. Otherwise, regions are often simpler to define. And it is easier to use nested items, as is explained in the next section.

44.5 Nested items

Take a look at this comment:

```
%Get input  TODO: Skip white space
```

You want to highlight TODO in big yellow letters, even though it is in a comment that is highlighted blue. To let Vim know about this, you define the following syntax groups:

```
:syntax keyword xTodo TODO contained
:syntax match xComment /%.*/ contains=xTodo
```

In the first line, the "contained" argument tells Vim that this keyword can exist only inside another syntax item. The next line has "contains=xTodo". This indicates that the xTodo syntax element is inside it. The result is that the comment line as a whole is matched with "xComment" and made blue. The word TODO inside it is matched by xTodo and highlighted yellow (highlighting for xTodo was setup for this).

RECURSIVE NESTING

The x language defines code blocks in curly braces. And a code block may contain other code blocks. This can be defined this way:

```
:syntax region xBlock start=/{/ end=/}/ contains=xBlock
```

Suppose you have this text:

```
while i < b {
if a {
b = c;
}
}
```

First a xBlock starts at the { in the first line. In the second line another { is found. Since we are inside a xBlock item, and it contains itself, a nested xBlock item will start here. Thus the "b = c" line is inside the second level xBlock region. Then a } is found in the next line, which matches with the end pattern of the region. This ends the nested xBlock. Because the } is included in the nested region, it is hidden from the first xBlock region. Then at the last } the first xBlock region ends.

KEEPING THE END

Consider the following two syntax items:

```
:syntax region xComment start=/%/ end=/$/ contained
:syntax region xPreProc start=/#/ end=/$/ contains=xComment
```

You define a comment as anything from % to the end of the line. A preprocessor directive is anything from # to the end of the line. Because you can have a comment on a preprocessor line, the preprocessor definition includes a "contains=xComment" argument. Now look what happens with this text:

```
#define X = Y  % Comment text
int foo = 1;
```

What you see is that the second line is also highlighted as xPreProc. The preprocessor directive should end at the end of the line. That is why you have used "end=/$/". So what is going wrong?

The problem is the contained comment. The comment starts with % and ends at the end of the line. After the comment ends, the preprocessor syntax continues. This is after the end of the line has been seen, so the next line is included as well.

To avoid this problem and to avoid a contained syntax item eating a needed end of line, use the "keepend" argument. This takes care of the double end-of-line matching:

```
:syntax region xComment start=/%/ end=/$/ contained
:syntax region xPreProc start=/#/ end=/$/ contains=xComment keepend
```

CONTAINING MANY ITEMS

You can use the contains argument to specify that everything can be contained.
For example:

```
:syntax region xList start=/\[/ end=/\]/ contains=ALL
```

All syntax items will be contained in this one. It also contains itself, but
not at the same position (that would cause an endless loop).
   You can specify that some groups are not contained. Thus contain all
groups but the ones that are listed:

```
:syntax region xList start=/\[/ end=/\]/ contains=ALLBUT,xString
```

With the "TOP" item you can include all items that don't have a "contained"
argument. "CONTAINED" is used to only include items with a "contained"
argument. See :syn-contains for the details.

44.6 Following groups

The x language has statements in this form:

if (condition) then

You want to highlight the three items differently. But "(condition)" and
"then" might also appear in other places, where they get different
highlighting. This is how you can do this:

```
:syntax match xIf /if/ nextgroup=xIfCondition skipwhite
:syntax match xIfCondition /([^)]*)/ contained nextgroup=xThen skipwhite
:syntax match xThen /then/ contained
```

The "nextgroup" argument specifies which item can come next. This is not
required. If none of the items that are specified are found, nothing happens.
For example, in this text:

if not (condition) then

The "if" is matched by xIf. "not" doesn't match the specified nextgroup
xIfCondition, thus only the "if" is highlighted.

The "skipwhite" argument tells Vim that white space (spaces and tabs) may
appear in between the items. Similar arguments are "skipnl", which allows a
line break in between the items, and "skipempty", which allows empty lines.
Notice that "skipnl" doesn't skip an empty line, something must match after
the line break.

44.7 Other arguments

MATCHGROUP

When you define a region, the entire region is highlighted according to the
group name specified. To highlight the text enclosed in parentheses () with
the group xInside, for example, use the following command:

```
:syntax region xInside start=/(/ end=/)/
```

Suppose, that you want to highlight the parentheses differently. You can do
this with a lot of convoluted region statements, or you can use the
"matchgroup" argument. This tells Vim to highlight the start and end of a
region with a different highlight group (in this case, the xParen group):

```
:syntax region xInside matchgroup=xParen start=/(/ end=/)/
```

The "matchgroup" argument applies to the start or end match that comes after
it. In the previous example both start and end are highlighted with xParen.
To highlight the end with xParenEnd:

```
:syntax region xInside matchgroup=xParen start=/(/
        \ matchgroup=xParenEnd end=/)/
```

A side effect of using "matchgroup" is that contained items will not match in
the start or end of the region. The example for "transparent" uses this.

TRANSPARENT

In a C language file you would like to highlight the () text after a "while"
differently from the () text after a "for". In both of these there can be
nested () items, which should be highlighted in the same way. You must make
sure the () highlighting stops at the matching). This is one way to do this:

```
:syntax region cWhile matchgroup=cWhile start=/while\s*(/ end=/)/
        \ contains=cCondNest
:syntax region cFor matchgroup=cFor start=/for\s*(/ end=/)/
        \ contains=cCondNest
:syntax region cCondNest start=/(/ end=/)/ contained transparent
```

Now you can give cWhile and cFor different highlighting. The cCondNest item
can appear in either of them, but take over the highlighting of the item it is
contained in. The "transparent" argument causes this.
   Notice that the "matchgroup" argument has the same group as the item
itself. Why define it then? Well, the side effect of using a matchgroup is
that contained items are not found in the match with the start item then.
This avoids that the cCondNest group matches the (just after the "while" or
"for". If this would happen, it would span the whole text until the matching
) and the region would continue after it. Now cCondNest only matches after
the match with the start pattern, thus after the first (.

OFFSETS

Suppose you want to define a region for the text between (and) after an
"if". But you don't want to include the "if" or the (and). You can do this
by specifying offsets for the patterns. Example:

```
:syntax region xCond start=/if\s*(/ms=e+1 end=/)/me=s-1
```

The offset for the start pattern is "ms=e+1". "ms" stands for Match Start.
This defines an offset for the start of the match. Normally the match starts
where the pattern matches. "e+1" means that the match now starts at the end
of the pattern match, and then one character further.
   The offset for the end pattern is "me=s-1". "me" stands for Match End.
"s-1" means the start of the pattern match and then one character back. The
result is that in this text:

if (foo == bar)

Only the text "foo == bar" will be highlighted as xCond.

More about offsets here: :syn-pattern-offset.

ONELINE

The "oneline" argument indicates that the region does not cross a line
boundary. For example:

```
:syntax region xIfThen start=/if/ end=/then/ oneline
```

This defines a region that starts at "if" and ends at "then". But if there is
no "then" after the "if", the region doesn't match.

> **Note:**
> When using "oneline" the region doesn't start if the end pattern
> doesn't match in the same line. Without "oneline" Vim does _not_
> check if there is a match for the end pattern. The region starts even
> when the end pattern doesn't match in the rest of the file.

CONTINUATION LINES AND AVOIDING THEM

Things now become a little more complex. Let's define a preprocessor line.
This starts with a # in the first column and continues until the end of the
line. A line that ends with \ makes the next line a continuation line. The
way you handle this is to allow the syntax item to contain a continuation
pattern:

```
:syntax region xPreProc start=/^#/ end=/$/ contains=xLineContinue
:syntax match xLineContinue "\\$" contained
```

In this case, although xPreProc normally matches a single line, the group
contained in it (namely xLineContinue) lets it go on for more than one line.
For example, it would match both of these lines:

**#define SPAM spam spam spam **
bacon and spam

In this case, this is what you want. If it is not what you want, you can call
for the region to be on a single line by adding "excludenl" to the contained
pattern. For example, you want to highlight "end" in xPreProc, but only at
the end of the line. To avoid making the xPreProc continue on the next line,
like xLineContinue does, use "excludenl" like this:

```
:syntax region xPreProc start=/^#/ end=/$/
        \ contains=xLineContinue,xPreProcEnd
:syntax match xPreProcEnd excludenl /end$/ contained
:syntax match xLineContinue "\\$" contained
```

"excludenl" must be placed before the pattern. Since "xLineContinue" doesn't
have "excludenl", a match with it will extend xPreProc to the next line as
before.

44.8 Clusters

One of the things you will notice as you start to write a syntax file is that
you wind up generating a lot of syntax groups. Vim enables you to define a
collection of syntax groups called a cluster.
   Suppose you have a language that contains for loops, if statements, while

loops, and functions. Each of them contains the same syntax elements: numbers
and identifiers. You define them like this:

```
:syntax match xFor /^for.*/ contains=xNumber,xIdent
:syntax match xIf /^if.*/ contains=xNumber,xIdent
:syntax match xWhile /^while.*/ contains=xNumber,xIdent
```

You have to repeat the same "contains=" every time. If you want to add
another contained item, you have to add it three times. Syntax clusters
simplify these definitions by enabling you to have one cluster stand for
several syntax groups.
   To define a cluster for the two items that the three groups contain, use
the following command:

```
:syntax cluster xState contains=xNumber,xIdent
```

Clusters are used inside other syntax items just like any syntax group.
Their names start with @. Thus, you can define the three groups like this:

```
:syntax match xFor /^for.*/ contains=@xState
:syntax match xIf /^if.*/ contains=@xState
:syntax match xWhile /^while.*/ contains=@xState
```

You can add new group names to this cluster with the "add" argument:

```
:syntax cluster xState add=xString
```

You can remove syntax groups from this list as well:

```
:syntax cluster xState remove=xNumber
```

44.9 Including another syntax file

The C++ language syntax is a superset of the C language. Because you do not
want to write two syntax files, you can have the C++ syntax file read in the
one for C by using the following command:

```
:runtime! syntax/c.vim
```

The ":runtime!" command searches 'runtimepath' for all "syntax/c.vim" files.
This makes the C parts of the C++ syntax be defined like for C files. If you
have replaced the c.vim syntax file, or added items with an extra file, these
will be loaded as well.
   After loading the C syntax items the specific C++ items can be defined.
For example, add keywords that are not used in C:

```
:syntax keyword cppStatement    new delete this friend using
```

This works just like in any other syntax file.

Now consider the Perl language. A Perl script consists of two distinct parts:
a documentation section in POD format, and a program written in Perl itself.
The POD section starts with "=head" and ends with "=cut".
   You want to define the POD syntax in one file, and use it from the Perl
syntax file. The ":syntax include" command reads in a syntax file and stores
the elements it defined in a syntax cluster. For Perl, the statements are as
follows:

```
:syntax include @Pod <script>:p:h/pod.vim
:syntax region perlPOD start=/^=head/ end=/^=cut/ contains=@Pod
```

When "=head" is found in a Perl file, the perlPOD region starts. In this
region the @Pod cluster is contained. All the items defined as top-level
items in the pod.vim syntax files will match here. When "=cut" is found, the
region ends and we go back to the items defined in the Perl file.
   The ":syntax include" command is clever enough to ignore a ":syntax clear"
command in the included file. And an argument such as "contains=ALL" will
only contain items defined in the included file, not in the file that includes
it.
   The "<script>:p:h/" part uses the name of the current file (<script>),
expands it to a full path (:p) and then takes the head (:h). This results in
the directory name of the file. This causes the pod.vim file in the same
directory to be included.

44.10 Synchronizing

Compilers have it easy. They start at the beginning of a file and parse it
straight through. Vim does not have it so easy. It must start in the middle,
where the editing is being done. So how does it tell where it is?
   The secret is the ":syntax sync" command. This tells Vim how to figure out
where it is. For example, the following command tells Vim to scan backward

for the beginning or end of a C-style comment and begin syntax coloring from
there:

```
:syntax sync ccomment
```

You can tune this processing with some arguments. The "minlines" argument
tells Vim the minimum number of lines to look backward, and "maxlines" tells
the editor the maximum number of lines to scan.
   For example, the following command tells Vim to look at least 10 lines
before the top of the screen:

```
:syntax sync ccomment minlines=10 maxlines=500
```

If it cannot figure out where it is in that space, it starts looking farther
and farther back until it figures out what to do. But it looks no farther
back than 500 lines. (A large "maxlines" slows down processing. A small one
might cause synchronization to fail.)
   To make synchronizing go a bit faster, tell Vim which syntax items can be
skipped. Every match and region that only needs to be used when actually
displaying text can be given the "display" argument.
   By default, the comment to be found will be colored as part of the Comment
syntax group. If you want to color things another way, you can specify a
different syntax group:

```
:syntax sync ccomment xAltComment
```

If your programming language does not have C-style comments in it, you can try
another method of synchronization. The simplest way is to tell Vim to space
back a number of lines and try to figure out things from there. The following
command tells Vim to go back 150 lines and start parsing from there:

```
:syntax sync minlines=150
```

A large "minlines" value can make Vim slower, especially when scrolling
backwards in the file.
   Finally, you can specify a syntax group to look for by using this command:

```
:syntax sync match {sync-group-name}
        \ grouphere {group-name} {pattern}
```

This tells Vim that when it sees {pattern} the syntax group named {group-name}
begins just after the pattern given. The {sync-group-name} is used to give a
name to this synchronization specification. For example, the sh scripting
language begins an if statement with "if" and ends it with "fi":

```
if [ --f file.txt ] ; then
echo "File exists"
fi
```

To define a "grouphere" directive for this syntax, you use the following
command:

```
:syntax sync match shIfSync grouphere shIf "\<if\>"
```

The "groupthere" argument tells Vim that the pattern ends a group. For
example, the end of the if/fi group is as follows:

```
:syntax sync match shIfSync groupthere NONE "\<fi\>"
```

In this example, the NONE tells Vim that you are not in any special syntax
region. In particular, you are not inside an if block.

You also can define matches and regions that are with no "grouphere" or
"groupthere" arguments. These groups are for syntax groups skipped during
synchronization. For example, the following skips over anything inside {},
even if it would normally match another synchronization method:

```
:syntax sync match xSpecial /{.*}/
```

More about synchronizing in the reference manual: :syn-sync.

44.11 Installing a syntax file

When your new syntax file is ready to be used, drop it in a "syntax" directory
in 'runtimepath'. For Unix that would be "~/.config/nvim/syntax".
  The name of the syntax file must be equal to the file type, with ".vim"
added. Thus for the x language, the full path of the file would be:

~/.config/nvim/syntax/x.vim

You must also make the file type be recognized. See 43.2.

If your file works well, you might want to make it available to other Vim
users. First read the next section to make sure your file works well for
others. Then e-mail it to the Vim maintainer: <maintainer@vim.org>. Also

explain how the filetype can be detected. With a bit of luck your file will
be included in the next Vim version!

ADDING TO AN EXISTING SYNTAX FILE

We were assuming you were adding a completely new syntax file. When an
existing syntax file works, but is missing some items, you can add items in a
separate file. That avoids changing the distributed syntax file, which will
be lost when installing a new version of Vim.
   Write syntax commands in your file, possibly using group names from the
existing syntax. For example, to add new variable types to the C syntax file:

```
:syntax keyword cType off_t uint
```

Write the file with the same name as the original syntax file. In this case
"c.vim". Place it in a directory near the end of 'runtimepath'. This makes
it loaded after the original syntax file. For Unix this would be:

~/.config/nvim/after/syntax/c.vim

44.12 Portable syntax file layout

Wouldn't it be nice if all Vim users exchange syntax files? To make this
possible, the syntax file must follow a few guidelines.

Start with a header that explains what the syntax file is for, who maintains
it and when it was last updated. Don't include too much information about
changes history, not many people will read it. Example:

```
" Vim syntax file
" Language:     C
" Maintainer:   Bram Moolenaar <Bram@vim.org>
" Last Change:  2001 Jun 18
" Remark:       Included by the C++ syntax.
```

Use the same layout as the other syntax files. Using an existing syntax file
as an example will save you a lot of time.

Choose a good, descriptive name for your syntax file. Use lowercase letters
and digits. Don't make it too long, it is used in many places: The name of
the syntax file "name.vim", 'filetype', b:current_syntax and the start of each
syntax group (nameType, nameStatement, nameString, etc).

Start with a check for "b:current_syntax". If it is defined, some other
syntax file, earlier in 'runtimepath' was already loaded:

```
if exists("b:current_syntax")
  finish
endif
```

Set "b:current_syntax" to the name of the syntax at the end. Don't forget
that included files do this too, you might have to reset "b:current_syntax" if
you include two files.

Do not include anything that is a user preference. Don't set 'tabstop',
'expandtab', etc. These belong in a filetype plugin.

Do not include mappings or abbreviations. Only include setting 'iskeyword' if
it is really necessary for recognizing keywords.

To allow users select their own preferred colors, make a different group name
for every kind of highlighted item. Then link each of them to one of the
standard highlight groups. That will make it work with every color scheme.
If you select specific colors it will look bad with some color schemes. And
don't forget that some people use a different background color, or have only
eight colors available.

For the linking use "hi def link", so that the user can select different
highlighting before your syntax file is loaded. Example:

```
hi def link nameString  String
hi def link nameNumber  Number
hi def link nameCommand Statement
... etc ...
```

Add the "display" argument to items that are not used when syncing, to speed
up scrolling backwards and CTRL-L.

Next chapter: usr_45.txt Select your language

Copyright: see manual-copyright

VIM USER MANUAL by Bram Moolenaar

Select your language (locale)

The messages in Vim can be given in several languages. This chapter explains how to change which one is used. Also, the different ways to work with files in various languages is explained.

45.1 Language for Messages
45.2 Language for Menus
45.3 Using another encoding
45.4 Editing files with a different encoding
45.5 Entering language text

Previous chapter: usr_44.txt Your own syntax highlighted
Table of contents: usr_toc.txt

Language for Messages

When you start Vim, it checks the environment to find out what language you are using. Mostly this should work fine, and you get the messages in your language (if they are available). To see what the current language is, use this command:

```
:language
```

If it replies with "C", this means the default is being used, which is English.

What if you would like your messages in a different language? There are several ways. Which one you should use depends on the capabilities of your system.
   The first way is to set the environment to the desired language before starting Vim. Example for Unix:

```
env LANG=de_DE.ISO_8859-1  vim
```

This only works if the language is available on your system. The advantage is that all the GUI messages and things in libraries will use the right language as well. A disadvantage is that you must do this before starting Vim. If you want to change language while Vim is running, you can use the second method:

```
:language fr_FR.ISO_8859-1
```

This way you can try out several names for your language. You will get an error message when it's not supported on your system. You don't get an error when translated messages are not available. Vim will silently fall back to using English.
   To find out which languages are supported on your system, find the directory where they are listed. On my system it is "/usr/share/locale". On some systems it's in "/usr/lib/locale". The manual page for "setlocale" should give you a hint where it is found on your system.
   Be careful to type the name exactly as it should be. Upper and lowercase matter, and the '-' and '_' characters are easily confused.

You can also set the language separately for messages, edited text and the time format. See :language.

DO-IT-YOURSELF MESSAGE TRANSLATION

If translated messages are not available for your language, you could write them yourself. To do this, get the source code for Vim and the GNU gettext package. After unpacking the sources, instructions can be found in the directory src/po/README.txt.
   It's not too difficult to do the translation. You don't need to be a programmer. You must know both English and the language you are translating to, of course.
   When you are satisfied with the translation, consider making it available to others. Upload it to https://github.com/vim/vim or e-mail it to the Vim maintainer <maintainer@vim.org>. Or both.

45.2 Language for Menus

The default menus are in English. To be able to use your local language, they must be translated. Normally this is automatically done for you if the environment is set for your language, just like with messages. You don't need to do anything extra for this. But it only works if translations for the language are available.
   Suppose you are in Germany, with the language set to German, but prefer to use "File" instead of "Datei". You can switch back to using the English menus this way:

```
:set langmenu=none
```

It is also possible to specify a language:

```
:set langmenu=nl_NL.ISO_8859-1
```

Like above, differences between "-" and "_" matter. However, upper/lowercase differences are ignored here.
   The 'langmenu' option must be set before the menus are loaded. Once the menus have been defined changing 'langmenu' has no direct effect. Therefore, put the command to set 'langmenu' in your vimrc file.
   If you really want to switch menu language while running Vim, you can do it this way:

```
:source $VIMRUNTIME/delmenu.vim
:set langmenu=de_DE.ISO_8859-1
:source $VIMRUNTIME/menu.vim
```

There is one drawback: All menus that you defined yourself will be gone. You will need to redefine them as well.

DO-IT-YOURSELF MENU TRANSLATION

To see which menu translations are available, look in this directory:

$VIMRUNTIME/lang

The files are called menu_{language}.vim. If you don't see the language you want to use, you can do your own translations. The simplest way to do this is by copying one of the existing language files, and change it.
   First find out the name of your language with the ":language" command. Use this name, but with all letters made lowercase. Then copy the file to your own runtime directory, as found early in 'runtimepath'. For example, for Unix you would do:

```
:!cp $VIMRUNTIME/lang/menu_ko_kr.euckr.vim ~/.config/nvim/lang/menu_nl_be.iso_8859-1.vim
```

You will find hints for the translation in "$VIMRUNTIME/lang/README.txt".

45.3 Using another encoding

Vim guesses that the files you are going to edit are encoded for your language. For many European languages this is "latin1". Then each byte is one character. That means there are 256 different characters possible. For Asian languages this is not sufficient. These mostly use a double-byte encoding, providing for over ten thousand possible characters. This still isn't enough when a text is to contain several different languages. This is where Unicode comes in. It was designed to include all characters used in commonly used languages. This is the "Super encoding that replaces all others". But it isn't used that much yet.
   Fortunately, Vim supports these three kinds of encodings. And, with some restrictions, you can use them even when your environment uses another language than the text.
   Nevertheless, when you only edit files that are in the encoding of your language, the default should work fine and you don't need to do anything. The following is only relevant when you want to edit different languages.

USING UNICODE IN THE GUI

The nice thing about Unicode is that other encodings can be converted to it and back without losing information. When you make Vim use Unicode internally, you will be able to edit files in any encoding.
   Unfortunately, the number of systems supporting Unicode is still limited. Thus it's unlikely that your language uses it. You need to tell Vim you want to use Unicode, and how to handle interfacing with the rest of the system.
   Let's start with the GUI version of Vim, which is able to display Unicode characters. This should work:

```
:set encoding=utf-8
:set guifont=-misc-fixed-medium-r-normal--18-120-100-100-c-90-iso10646-1
```

The 'encoding' option tells Vim the encoding of the characters that you use. This applies to the text in buffers (files you are editing), registers, Vim script files, etc. You can regard 'encoding' as the setting for the internals of Vim.
   This example assumes you have this font on your system. The name in the example is for the X Window System. This font is in a package that is used to enhance xterm with Unicode support.

For MS-Windows, some fonts have a limited number of Unicode characters. Try using the "Courier New" font. You can use the Edit/Select Font... menu to select and try out the fonts available. Only fixed-width fonts can be used though. Example:

```
:set guifont=courier_new:h12
```

If it doesn't work well, try getting a fontpack.

Now you have told Vim to use Unicode internally and display text with a Unicode font.

USING UNICODE IN A UNICODE TERMINAL

There are terminals that support Unicode directly. The standard xterm that comes with XFree86 is one of them. Let's use that as an example.
First of all, the xterm must have been compiled with Unicode support.
Start the xterm with the "-u8" argument. You might also need so specify a font. Example:

```
xterm -u8 -fn -misc-fixed-medium-r-normal--18-120-100-100-c-90-iso10646-1
```

Now you can run Vim inside this terminal.

USING UNICODE IN AN ORDINARY TERMINAL

Suppose you want to work with Unicode files, but don't have a terminal with Unicode support. You can do this with Vim, although characters that are not supported by the terminal will not be displayed. The layout of the text will be preserved.

Try editing a file with Unicode characters in it. You will notice that Vim will put a question mark (or underscore or some other character) in places where a character should be that the terminal can't display. Move the cursor to a question mark and use this command:

```
ga
```

Vim will display a line with the code of the character. This gives you a hint about what character it is. You can look it up in a Unicode table. You could actually view a file that way, if you have lots of time at hand.

> **Note:**
> Since 'encoding' is used for all text inside Vim, changing it makes all non-ASCII text invalid. You will notice this when using registers and the shada-file (e.g., a remembered search pattern). It's recommended to set 'encoding' in your vimrc file, and leave it alone.

45.4 Editing files with a different encoding

Suppose you have setup Vim to use Unicode, and you want to edit a file that is in 16-bit Unicode. Sounds simple, right? Well, Vim actually uses utf-8 encoding internally, thus the 16-bit encoding must be converted, since there is a difference between the character set (Unicode) and the encoding (utf-8 or 16-bit).
Vim will try to detect what kind of file you are editing. It uses the encoding names in the 'fileencodings' option. When using Unicode, the default value is: "ucs-bom,utf-8,latin1". This means that Vim checks the file to see if it's one of these encodings:

ucs-bom	File must start with a Byte Order Mark (BOM). This allows detection of 16-bit, 32-bit and utf-8 Unicode encodings.
utf-8	utf-8 Unicode. This is rejected when a sequence of bytes is illegal in utf-8.
latin1	The good old 8-bit encoding. Always works.

When you start editing that 16-bit Unicode file, and it has a BOM, Vim will detect this and convert the file to utf-8 when reading it. The 'fileencoding' option (without s at the end) is set to the detected value. In this case it is "utf-16le". That means it's Unicode, 16-bit and little-endian. This file format is common on MS-Windows (e.g., for registry files).
When writing the file, Vim will compare 'fileencoding' with 'encoding'. If they are different, the text will be converted.
An empty value for 'fileencoding' means that no conversion is to be done. Thus the text is assumed to be encoded with 'encoding'.

If the default 'fileencodings' value is not good for you, set it to the encodings you want Vim to try. Only when a value is found to be invalid will the next one be used. Putting "latin1" first doesn't work, because it is never illegal. An example, to fall back to Japanese when the file doesn't have a BOM and isn't utf-8:

```
:set fileencodings=ucs-bom,utf-8,sjis
```

See encoding-values for suggested values. Other values may work as well. This depends on the conversion available.

FORCING AN ENCODING

If the automatic detection doesn't work you must tell Vim what encoding the file is. Example:

```
:edit ++enc=koi8-r russian.txt
```

The "++enc" part specifies the name of the encoding to be used for this file only. Vim will convert the file from the specified encoding, Russian in this example, to 'encoding'. 'fileencoding' will also be set to the specified encoding, so that the reverse conversion can be done when writing the file.
The same argument can be used when writing the file. This way you can actually use Vim to convert a file. Example:

```
:write ++enc=utf-8 russian.txt
```

Note:
Conversion may result in lost characters. Conversion from an encoding to Unicode and back is mostly free of this problem, unless there are illegal characters. Conversion from Unicode to other encodings often loses information when there was more than one language in the file.

45.5 Entering language text

Computer keyboards don't have much more than a hundred keys. Some languages have thousands of characters, Unicode has over hundred thousand. So how do you type these characters?

First of all, when you don't use too many of the special characters, you can use digraphs. This was already explained in 24.9.

When you use a language that uses many more characters than keys on your keyboard, you will want to use an Input Method (IM). This requires learning the translation from typed keys to resulting character. When you need an IM you probably already have one on your system. It should work with Vim like with other programs.

KEYMAPS

For some languages the character set is different from latin, but uses a similar number of characters. It's possible to map keys to characters. Vim uses keymaps for this.

Suppose you want to type Hebrew. You can load the keymap like this:

```
:set keymap=hebrew
```

Vim will try to find a keymap file for you. This depends on the value of 'encoding'. If no matching file was found, you will get an error message.

Now you can type Hebrew in Insert mode. In Normal mode, and when typing a ":" command, Vim automatically switches to English. You can use this command to switch between Hebrew and English:

```
CTRL-^
```

This only works in Insert mode and Command-line mode. In Normal mode it does something completely different (jumps to alternate file).

The usage of the keymap is indicated in the mode message, if you have the 'showmode' option set. In the GUI Vim will indicate the usage of keymaps with a different cursor color.

You can also change the usage of the keymap with the 'iminsert' and 'imsearch' options.

To see the list of mappings, use this command:

```
:lmap
```

To find out which keymap files are available, in the GUI you can use the Edit/Keymap menu. Otherwise you can use this command:

```
:echo globpath(&rtp, "keymap/*.vim")
```

DO-IT-YOURSELF KEYMAPS

You can create your own keymap file. It's not very difficult. Start with a keymap file that is similar to the language you want to use. Copy it to the "keymap" directory in your runtime directory. For example, for Unix, you would use the directory "~/.config/nvim/keymap".

The name of the keymap file must look like this:

keymap/{name}.vim
or
keymap/{name}_{encoding}.vim

{name} is the name of the keymap. Chose a name that is obvious, but different from existing keymaps (unless you want to replace an existing keymap file). {name} cannot contain an underscore. Optionally, add the encoding used after an underscore. Examples:

keymap/hebrew.vim
keymap/hebrew_utf-8.vim

The contents of the file should be self-explanatory. Look at a few of the keymaps that are distributed with Vim. For the details, see mbyte-keymap.

LAST RESORT

If all other methods fail, you can enter any character with CTRL-V:

encoding type	range	
8-bit	CTRL-V 123	decimal 0-255
8-bit	CTRL-V x a1	hexadecimal 00-ff
16-bit CTRL-V u 013b	hexadecimal 0000-ffff	
31-bit	CTRL-V U 001303a4	hexadecimal 00000000-7fffffff

Don't type the spaces. See i_CTRL-V_digit for the details.

Copyright: see manual-copyright

This file contains an alphabetical list of messages and error messages that
Vim produces. You can use this if you don't understand what the message
means. It is not complete though.

Messages history :mes message-history

The ":messages" command can be used to view previously given messages. This
is especially useful when messages have been overwritten or truncated. This
depends on the 'shortmess' option.

:mes[sages]	Show all messages.
:{count}mes[sages]	Show the {count} most recent messages.
:mes[sages] clear	Clear all messages.
:{count}mes[sages] clear	Clear messages, keeping only the {count} most recent ones.

The number of remembered messages is determined by the 'messagesopt' option.

g<

The "g<" command shows the last page of previous command output.
This only works when 'more' is set.
With ui2: "g<" shows recent messages in the pager.

If you want to find help on a specific (error) message, use the ID at the
start of the message. For example, to get help on the message:

```
E72: Close error on swap file
```

or (translated):

```
E72: Errore durante chiusura swap file
```

Use:

```
:help E72
```

If you are lazy, it also works without the shift key:

```
:help e72
```

Error messages error-messages errors

When an error message is displayed, but it is removed before you could read
it, you can see it again with:

```
:echo v:errmsg
```

Or view a list of recent messages with:

```
:messages
```

See :messages above.

LIST OF MESSAGES

E222 E228 E232 E293 E298 E304 E317
E318 E356 E438 E439 E440 E316 E320 E322
E323 E341 E473 E570 E685 E292

```
Add to read buffer
makemap: Illegal mode
Cannot create BalloonEval with both message and callback
Hangul automata ERROR
block was not locked
Didn't get block nr {N}?
ml_upd_block0(): Didn't get block 0??
pointer block id wrong {N}
Updated too many blocks?
get_varp ERROR
u_undo: line numbers wrong
undo list corrupt
undo line missing
ml_get: cannot find line {N}
cannot find line {N}
line number out of range: {N} past the end
line count wrong in block {N}
Internal error
Internal error: {function}
fatal error in cs_manage_matches
Invalid count for del_bytes(): {N}
```

This is an internal error. If you can reproduce it, please send in a bug
report. bugs

```
ATTENTION
Found a swap file by the name ...
```

See ATTENTION.

E92

```
Buffer {N} not found
```

The buffer you requested does not exist. This can also happen when you have
wiped out a buffer which contains a mark or is referenced in another way.
:bwipeout

E95

```
Buffer with this name already exists
```

You cannot have two buffers with exactly the same name. This includes the
path leading to the file.

E1513

```
Cannot switch buffer. 'winfixbuf' is enabled
```

If a window has 'winfixbuf' enabled, you cannot change that window's current
buffer. You need to set 'nowinfixbuf' before continuing. You may use [!] to
force the window to switch buffers, if your command supports it.

E72

```
Close error on swap file
```

The swap-file, that is used to keep a copy of the edited text, could not be
closed properly. Mostly harmless.

E169

```
Command too recursive
```

This happens when an Ex command executes an Ex command that executes an Ex
command, etc. The limit is 200 or the value of 'maxfuncdepth', whatever is
larger. When it's more there probably is an endless loop. Probably a
:execute or :source command is involved.

E254

```
Cannot allocate color {name}
```

The color name {name} is unknown. See gui-colors for a list of colors that
are available on most systems.

E458

```
Cannot allocate colormap entry, some colors may be incorrect
```

This means that there are not enough colors available for Vim. It will still
run, but some of the colors will not appear in the specified color. Try
stopping other applications that use many colors, or start them after starting
gvim.
Browsers are known to consume a lot of colors. You can avoid this with
netscape by telling it to use its own colormap:

```
netscape -install
```

Or tell it to limit to a certain number of colors (64 should work well):

```
netscape -ncols 64
```

This can also be done with a line in your Xdefaults file:

```
Netscape*installColormap: Yes
```

or

```
Netscape*maxImageColors:  64
```

E79

```
Cannot expand wildcards
```

A filename contains a strange combination of characters, which causes Vim to attempt expanding wildcards but this fails. This does NOT mean that no matching file names could be found, but that the pattern was illegal.

E459

```
Cannot go back to previous directory
```

While expanding a file name, Vim failed to go back to the previously used directory. All file names being used may be invalid now! You need to have execute permission on the current directory.

E190 E212

```
Cannot open "{filename}" for writing
Can't open file for writing
```

For some reason the file you are writing to cannot be created or overwritten. The reason could be that you do not have permission to write in the directory or the file name is not valid.

E166

```
Can't open linked file for writing
```

You are trying to write to a file which can't be overwritten, and the file is a link (either a hard link or a symbolic link). Writing might still be possible if the directory that contains the link or the file is writable, but Vim now doesn't know if you want to delete the link and write the file in its place, or if you want to delete the file itself and write the new file in its place. If you really want to write the file under this name, you have to manually delete the link or the file, or change the permissions so that Vim can overwrite.

E46

```
Cannot change read-only variable "{name}"
```

You are trying to assign a value to an argument of a function a:var or a Vim internal variable v:var which is read-only.

E90

```
Cannot unload last buffer
```

Vim always requires one buffer to be loaded, otherwise there would be nothing to display in the window.

E40

```
Can't open errorfile <filename>
```

When using the ":make" or ":grep" commands: The file used to save the error messages or grep output cannot be opened. This can have several causes:

- 'shellredir' has a wrong value.
- The shell changes directory, causing the error file to be written in another directory. This could be fixed by changing 'makeef', but then the make command is still executed in the wrong directory.
- 'makeef' has a wrong value.
- The 'grepprg' or 'makeprg' could not be executed. This cannot always be detected (especially on MS-Windows). Check your $PATH.

```
Can't open file C:\TEMP\VIoD243.TMP
```

On MS-Windows, this message appears when the output of an external command was to be read, but the command didn't run successfully. This can be caused by many things. Check the 'shell', 'shellquote', 'shellxquote', 'shellslash' and related options. It might also be that the external command was not found, there is no different error message for that.

E12

```
Command not allowed from exrc/vimrc in current dir or tag search
```

Some commands are not allowed for security reasons. These commands mostly
come from a .exrc or .nvimrc file in the current directory, or from a tags
file. Also see 'secure'.

E74

```
Command too complex
```

A mapping resulted in a very long command string. Could be caused by a
mapping that indirectly calls itself.

```
CONVERSION ERROR
```

When writing a file and the text "CONVERSION ERROR" appears, this means that
some bits were lost when converting text from the internally used UTF-8 to the
format of the file. The file will not be marked unmodified. If you care
about the loss of information, set the 'fileencoding' option to another value
that can handle the characters in the buffer and write again. If you don't
care, you can abandon the buffer or reset the 'modified' option.
If there is a backup file, when 'writebackup' or 'backup' is set, it will not
be deleted, so you can move it back into place if you want to discard the
changes.

E302

```
Could not rename swap file
```

When the file name changes, Vim tries to rename the swap-file as well.
This failed and the old swap file is now still used. Mostly harmless.

E43 E44

```
Damaged match string
Corrupted regexp program
```

Something inside Vim went wrong and resulted in a corrupted regexp. If you
know how to reproduce this problem, please report it. bugs

E208 E209 E210

```
Error writing to "{filename}"
Error closing "{filename}"
Error reading "{filename}"
```

This occurs when Vim is trying to rename a file, but a simple change of file
name doesn't work. Then the file will be copied, but somehow this failed.
The result may be that both the original file and the destination file exist
and the destination file may be incomplete.

```
Vim: Error reading input, exiting...
```

This occurs when Vim cannot read typed characters while input is required.
Vim got stuck, the only thing it can do is exit. This can happen when both
stdin and stderr are redirected and executing a script that doesn't exit Vim.

E47

```
Error while reading errorfile
```

Reading the error file was not possible. This is NOT caused by an error
message that was not recognized.

E80

```
Error while writing
```

Writing a file was not completed successfully. The file is probably
incomplete.

E13 E189

```
File exists (add ! to override)
"{filename}" exists (add ! to override)
```

You are protected from accidentally overwriting a file. When you want to
write anyway, use the same command, but add a "!" just after the command.
Example:

```
:w /tmp/test
```

changes to:

```
:w! /tmp/test
```

E768

```
Swap file exists: {filename} (:silent! overrides)
```

You are protected from overwriting a file that is being edited by Vim. This
happens when you use ":w! filename" and a swapfile is found.

- If the swapfile was left over from an old crashed edit session you may want

  to delete the swapfile. Edit {filename} to find out information about the
  swapfile.
- If you want to write anyway prepend ":silent!" to the command. For example:

  ```
  :silent! w! /tmp/test
  ```

  The special command is needed, since you already added the ! for overwriting
  an existing file.

E139

```
File is loaded in another buffer
```

You are trying to write a file under a name which is also used in another
buffer. This would result in two versions of the same file.

E142

```
File not written: Writing is disabled by 'write' option
```

The 'write' option is off. This makes all commands that try to write a file
generate this message. This could be caused by a -m commandline argument.
You can switch the 'write' option on with ":set write".

E25

```
Nvim does not have a built-in GUI
```

Nvim does not have a built in GUI, so :gvim and :gui don't work.

E49

```
Invalid scroll size
```

This is caused by setting an invalid value for the 'scroll', 'scrolljump' or
'scrolloff' options.

E17

```
"{filename}" is a directory
```

You tried to write a file with the name of a directory. This is not possible.
You probably need to append a file name.

E19

```
Mark has invalid line number
```

You are using a mark that has a line number that doesn't exist. This can
happen when you have a mark in another file, and some other program has
deleted lines from it.

E219 E220

```
Missing {.
Missing }.
```

Using a {} construct in a file name, but there is a { without a matching } or
the other way around. It should be used like this: {foo,bar}. This matches
"foo" and "bar".

E315

```
ml_get: invalid lnum: {number}
```

This is an internal Vim error. Please try to find out how it can be
reproduced, and submit a bug-report.

E173

```
{number} more files to edit
```

You are trying to exit, while the last item in the argument list has not been

edited. This protects you from accidentally exiting when you still have more files to work on. See argument-list. If you do want to exit, just do it again and it will work.

E23 E194

```
No alternate file
No alternate file name to substitute for '#'
```

The alternate file is not defined yet. See alternate-file.

E32

```
No file name
```

The current buffer has no name. To write it, use ":w fname". Or give the buffer a name with ":file fname".

E141

```
No file name for buffer {number}
```

One of the buffers that was changed does not have a file name. Therefore it cannot be written. You need to give the buffer a file name:

```
:buffer {number}
:file {filename}
```

E33

```
No previous substitute regular expression
```

When using the '~' character in a pattern, it is replaced with the previously used pattern in a ":substitute" command. This fails when no such command has been used yet. See /~. This also happens when using ":s/pat/%/", where the "%" stands for the previous substitute string.

E35

```
No previous regular expression
```

When using an empty search pattern, the previous search pattern is used. But that is not possible if there was no previous search.

E24

```
No such abbreviation
```

You have used an ":unabbreviate" command with an argument which is not an existing abbreviation. All variations of this command give the same message: ":cunabbrev", ":iunabbrev", etc. Check for trailing white space.

E31

```
No such mapping
```

You have used an ":unmap" command with an argument which is not an existing mapping. All variations of this command give the same message: ":cunmap", ":unmap!", etc. A few hints:

- Check for trailing white space.

- If the mapping is buffer-local you need to use ":unmap <buffer>".

:map-<buffer>

E37 E89

```
No write since last change (add ! to override)
No write since last change for buffer {N} (add ! to override)
```

You are trying to abandon a file that has changes. Vim protects you from losing your work. You can either write the changed file with ":w", or, if you are sure, abandon it anyway, and lose all the changes. This can be done by adding a '!' character just after the command you used. Example:

```
:e other_file
```

changes to:

```
:e! other_file
```

E162

```
No write since last change for buffer "{name}"
```

This appears when you try to exit Vim while some buffers are changed. You will either have to write the changed buffer (with :w), or use a command to abandon the buffer forcefully, e.g., with ":qa!". Careful, make sure you don't throw away changes you really want to keep. You might have forgotten about a buffer, especially when 'hidden' is set.

```
[No write since last change]
```

This appears when executing a shell command while at least one buffer was changed. To avoid the message reset the 'warn' option.

E38

```
Null argument
```

Something inside Vim went wrong and resulted in a NULL pointer. If you know how to reproduce this problem, please report it. bugs

E41 E82 E83 E342

```
Out of memory!
Out of memory!  (allocating {number} bytes)
Cannot allocate any buffer, exiting...
Cannot allocate buffer, using other one...
```

Oh, oh. You must have been doing something complicated, or some other program is consuming your memory. Be careful! Vim is not completely prepared for an out-of-memory situation. First make sure that any changes are saved. Then try to solve the memory shortage. To stay on the safe side, exit Vim and start again.

Buffers are only partly kept in memory, thus editing a very large file is unlikely to cause an out-of-memory situation. Undo information is completely in memory, you can reduce that with these options:

- 'undolevels' Set to a low value, or to -1 to disable undo completely. This helps for a change that affects all lines.
- 'undoreload' Set to zero to disable.

E339

```
Pattern too long
```

This happens on systems with 16 bit ints: The compiled regexp pattern is longer than about 65000 characters. Try using a shorter pattern.
It also happens when the offset of a rule doesn't fit in the space available.
Try simplifying the pattern.

E45

```
'readonly' option is set (add ! to override)
```

You are trying to write a file that was marked as read-only. To write the file anyway, either reset the 'readonly' option, or add a '!' character just after the command you used. Example:

```
:w
```

changes to:

```
:w!
```

E294 E295 E301

```
Read error in swap file
Seek error in swap file read
Oops, lost the swap file!!!
```

Vim tried to read text from the swap-file, but something went wrong. The text in the related buffer may now be corrupted! Check carefully before you write a buffer. You may want to write it in another file and check for differences.

E192

```
Recursive use of :normal too deep
```

You are using a ":normal" command, whose argument again uses a ":normal"

command in a recursive way. This is restricted to 'maxmapdepth' levels. This
example illustrates how to get this message:

```
:map gq :normal gq<CR>
```

If you type "gq", it will execute this mapping, which will call "gq" again.

E22

```
Scripts nested too deep
```

Scripts can be read with the "-s" command-line argument and with the
:source! command. The script can then again read another script. This can
continue for about 14 levels. When more nesting is done, Vim assumes that
there is a recursive loop and stops with this error message.

E300

```
Swap file already exists (symlink attack?)
```

This message appears when Vim is trying to open a swap file and finds it
already exists or finds a symbolic link in its place. This shouldn't happen,
because Vim already checked that the file doesn't exist. Either someone else
opened the same file at exactly the same moment (very unlikely) or someone is
attempting a symlink attack (could happen when editing a file in /tmp or when
'directory' starts with "/tmp", which is a bad choice).

E432

```
Tags file not sorted: {file name}
```

Vim (and Vi) expect tags files to be sorted in ASCII order. Binary searching
can then be used, which is a lot faster than a linear search. If your tags
files are not properly sorted, reset the 'tagbsearch' option.
This message is only given when Vim detects a problem when searching for a
tag. Sometimes this message is not given, even though the tags file is not
properly sorted.

E424

```
Too many different highlighting attributes in use
```

Vim can only handle about 223 different kinds of highlighting. If you run
into this limit, you have used too many :highlight commands with different
arguments. A ":highlight link" is not counted.

E77

```
Too many file names (glob not allowed)
```

When expanding file names, more than one match was found. Only one match is
allowed for the command that was used.
Use :next if you want to edit all files matching a glob pattern.

E303

```
Unable to open swap file for "{filename}", recovery impossible
```

Vim was not able to create a swap file. You can still edit the file, but if
Vim unexpectedly exits the changes will be lost. And Vim may consume a lot of
memory when editing a big file. You may want to change the 'directory' option
to avoid this error. This error is not given when 'directory' is empty. See
swap-file.

E140

```
Use ! to write partial buffer
```

When using a range to write part of a buffer, it is unusual to overwrite the
original file. It is probably a mistake (e.g., when Visual mode was active
when using ":w"), therefore Vim requires using a ! after the command, e.g.:
":3,10w!".

```
Warning: Cannot convert string "<Key>Escape,_Key_Cancel" to type
VirtualBinding
```

Messages like this appear when starting up. This is not a Vim problem, your
X11 configuration is wrong.

W10

```
Warning: Changing a readonly file
```

The file is read-only and you are making a change to it anyway. You can use

the FileChangedRO autocommand event to avoid this message (the autocommand must reset the 'readonly' option). See 'modifiable' to completely disallow making changes to a file.
This message is only given for the first change after 'readonly' has been set.

W13

```
Warning: File "{filename}" has been created after editing started
```

You are editing a file in Vim when it didn't exist, but it does exist now. You will have to decide if you want to keep the version in Vim or the newly created file. This message is not given when 'buftype' is not empty.

W11

```
Warning: File "{filename}" has changed since editing started
```

The file which you have started editing has got another timestamp and the contents changed (more precisely: When reading the file again with the current option settings and autocommands you would end up with different text). This probably means that some other program changed the file. You will have to find out what happened, and decide which version of the file you want to keep. Set the 'autoread' option if you want to do this automatically.
This message is not given when 'buftype' is not empty.
Also see the FileChangedShell autocommand.

You will be given a dialog with the following options:

"OK": Dismiss the warning and continue editing. No changes are loaded, the buffer remains as it is.

"Load File": Reload the file from disk, replacing the current buffer contents. Any changes you made in Vim that haven't been saved will be lost.

"Load File and Options": Reload the file from disk and, in addition, apply relevant file settings, such as indentation, syntax highlighting, text width, and other filetype-specific options. This ensures the buffer matches the file's intended configuration according to your current settings and autocommands.

There is one situation where you get this message even though there is nothing wrong: If you save a file in Windows on the day the daylight saving time starts. It can be fixed in one of these ways:

- Add this line in your autoexec.bat:

  ```
  SET TZ=-1
  ```

  Adjust the "-1" for your time zone.
- Disable "automatically adjust clock for daylight saving changes".

- Just write the file again the next day. Or set your clock to the next day,

  write the file twice and set the clock back.

If you get W11 all the time, you may need to disable "Acronis Active Protection" or register Vim as a trusted service/application.

W12

```
Warning: File "{filename}" has changed and the buffer was changed in Vim as well
```

Like the above, and the buffer for the file was changed in this Vim as well. You will have to decide if you want to keep the version in this Vim or the one on disk. This message is not given when 'buftype' is not empty.

W16

```
Warning: Mode of file "{filename}" has changed since editing started
```

When the timestamp for a buffer was changed and the contents are still the same but the mode (permissions) have changed. This usually occurs when checking out a file from a version control system, which causes the read-only bit to be reset. It should be safe to reload the file. Set 'autoread' to automatically reload the file.

E211

```
File "{filename}" no longer available
```

The file which you have started editing has disappeared, or is no longer accessible. Make sure you write the buffer somewhere to avoid losing changes. This message is not given when 'buftype' is not empty.

W14

```
Warning: List of file names overflow
```

You must be using an awful lot of buffers. It's now possible that two buffers
have the same number, which causes various problems. You might want to exit
Vim and restart it.

E931

```
Buffer cannot be registered
```

Out of memory or a duplicate buffer number. May happen after W14. Looking up
a buffer will not always work, better restart Vim.

E296 E297

```
Seek error in swap file write
Write error in swap file
```

This mostly happens when the disk is full. Vim could not write text into the
swap-file. It's not directly harmful, but when Vim unexpectedly exits some
text may be lost without recovery being possible. Vim might run out of memory
when this problem persists.

E10

```
\\ should be followed by /, ? or &
```

A command line started with a backslash or the range of a command contained a
backslash in a wrong place. This is often caused by command-line continuation
being disabled. Remove the 'C' flag from the 'cpoptions' option to enable it.

E471

```
Argument required
```

Ex command was executed without a mandatory argument(s).

E474 E475 E983

```
Invalid argument
Invalid argument: {arg}
Duplicate argument: {arg}
```

Ex command or function was given an invalid argument. Or jobstart() or
system() was given a non-executable command.

E488

```
Trailing characters
Trailing characters: {text}
```

An argument was given to an Ex command that does not permit one.
Or the argument has invalid characters and has not been recognized.

E477 E478

```
No ! allowed
Don't panic!
```

You have added a "!" after an Ex command that doesn't permit one.

E481

```
No range allowed
```

A range was specified for an Ex command that doesn't permit one. See
cmdline-ranges.

E482 E483

```
Can't create file {filename}
Can't get temp file name
```

Vim cannot create a temporary file.

E484 E485

```
Can't open file {filename}
Can't read file {filename}
```

Vim cannot read a temporary file. Especially on Windows, this can be caused
by wrong escaping of special characters for cmd.exe; the approach was
changed with patch 7.3.443. Try using shellescape() for all shell arguments
given to system(), or explicitly add escaping with ^. Also see
'shellxquote' and 'shellxescape'.

E464

```
Ambiguous use of user-defined command
```

There are two user-defined commands with a common name prefix, and you used
Command-line completion to execute one of them. user-cmd-ambiguous
Example:

```
:command MyCommand1 echo "one"
:command MyCommand2 echo "two"
:MyCommand
```

E492

```
Not an editor command
```

You tried to execute a command that is neither an Ex command nor
a user-defined command.

E905

```
Cannot set this option after startup
```

You tried to set an option after startup that only allows changes during
startup.

E943

```
Command table needs to be updated, run 'make'
```

This can only happen when changing the source code, after adding a command in
src/ex_cmds.lua. Update the lookup table by re-running the build.

E970

```
Failed to initialize lua interpreter
Failed to initialize builtin lua modules
```

Nvim failed to initialize the Lua interpreter or its builtin modules during
startup. This is a fatal error that prevents Nvim from running.

E5107

```
Lua: {error}
```

Lua code compilation (loadbuffer) failed when executing :lua commands or
functions. {error} contains the Lua syntax or compilation error message.

E5108

```
Lua function: {error}
Lua: {error}
```

Lua code execution (pcall) failed, when executing Lua code or during Lua
function completion. {error} contains the Lua runtime error message and
traceback.

E5109

```
Lua: {error}
```

Lua code compilation (loadbuffer) failed during :luado. {error} contains the
Lua syntax or compilation error message.

E5110

```
Lua: {error}
```

Lua code execution (pcall) failed during :luado command initialization.
{error} contains the Lua runtime error message and traceback.

E5111

```
Lua: {error}
```

Lua code execution failed in :luado callback or during Lua script loading.
{error} contains the Lua runtime error message and traceback.

E5112

```
Lua chunk: {error}
```

Lua chunk compilation failed. The loadstring() function returned an error
when trying to compile the Lua code. {error} contains the compilation error
message.

E5113

```
Lua chunk: {error}
```

Lua chunk execution failed after successful compilation. {error} contains the
Lua runtime error message and traceback.

E5114

```
Converting print argument #{i}: {error}
```

Lua print() failed to convert argument number {i} to a string. {error}
contains the conversion error details.

E5115

```
Loading Lua debug string: {error}
```

Failed to load (compile) a Lua debug command string. {error} contains the
compilation error message.

E5116

```
Calling Lua debug string: {error}
```

Failed to execute a Lua debug command string. {error} contains the runtime
error message and traceback.

E5117

```
Lua mapping no longer exists
```

Replayed keys (., CmdAtom keys, or keys captured by vim.on_key())
referenced a Lua-callback mapping that was deleted or redefined since.

Messages

messages

Messages are shown in the ui2 messages area. If they exceed ("overflow") the
area then they will be collapsed and a "spill" indicator will indicate how
many lines were collapsed. Use g< to see the full messages.

To avoid overflow ("spill") of the ui2 messages area:

- Set 'cmdheight' to 2 or higher.
- Add flags to 'shortmess'.
- Reset 'showcmd' and/or 'ruler'.

more-prompt pager

```
f/d/j: screen/page/line down, b/u/k: up, <Esc>: stop paging
```

This message is given when the ui2 modal dialog is active.
Only when the 'more' option is on.
Highlighted with the hl-MoreMsg group.

With ui2: use "g<" to show messages in the pager. When a command emits a
message while a previous message is still displayed, the messages are
collected in the pager window without focusing it: type "g<" to focus it, or
any other typed key to dismiss it.

Type	Effect
<CR> or <NL> or j or <Down>	one more line
d	down a page (half a screen)
<Space> or f or <PageDown> or CTRL-F	down a screen
G	down all the way, until the hit-enter prompt

```
    <BS> or k or <Up>                   one line back
    u                                   up a page (half a screen)
    b or <PageUp> or CTRL-B             back a screen
    g                                   back to the start

    q, <Esc> or CTRL-C                  stop the listing
    :                                   stop the listing and enter a
                                        command-line

   <C-Y>                                yank (copy) a modeless
                                        selection to the clipboard
                                        ("* and "+ registers)

   {menu-entry}                         what the menu is defined to
                                        in Cmdline-mode.

   <LeftMouse>                          next page*
```

Progress messages

progress-message

Plugins and core Nvim features can emit "progress" ui-messages to report the state of long-running tasks.

Create or update a progress-message by calling nvim_echo() with kind='progress'. Each message has a unique message-id, returned by nvim_echo. You can specify the id when calling nvim_echo to update an existing progress-message.

Note: To avoid a "dangling" progress-message, always emit a non-"running" status, when your task is done; otherwise consumers (including Nvim itself) tracking the task by its message-id cannot know when the task is done.

Events:

- msg_show ui-messages event is fired for ext-ui upon creation/update of a progress-message
- Updating or creating a progress message also triggers the Progress autocommand.

Example:

```
local progress = {
  kind = 'progress',
  percent = 10,
  source = 'my-plugin',
  status = 'running',
  title = 'term',
}
progress.id = vim.api.nvim_echo({ { 'searching...' } }, true, progress)
progress.percent = 50
vim.api.nvim_echo({ { 'searching' } }, true, progress)
progress.status = 'success'
progress.percent = 100
vim.api.nvim_echo({ { 'done' } }, true, progress)
```

See also: nvim_echo() ui-messages Progress

Cursor motions navigation

These commands move the cursor position. If the new position is off of the screen, the screen is scrolled to show the cursor (see also 'scrolljump' and 'scrolloff' options).

General remarks:

If you want to know where you are in the file use the "CTRL-G" command CTRL-G or the "g CTRL-G" command g_CTRL-G. If you set the 'ruler' option, the cursor position is continuously shown in the status line (which slows down Vim a little).

Experienced users prefer the hjkl keys because they are always right under their fingers. Beginners often prefer the arrow keys, because they do not know what the hjkl keys do. The mnemonic value of hjkl is clear from looking at the keyboard. Think of j as an arrow pointing downwards.

The 'virtualedit' option can be set to make it possible to move the cursor to positions where there is no character or within a multi-column character (like a tab).

1. Motions and operators operator

The motion commands can be used after an operator command, to have the command operate on the text that was moved over. That is the text between the cursor position before and after the motion. Operators are generally used to delete or change text. The following operators are available:

c	c	change
d	d	delete
y	y	yank into register (does not change the text)
~	~	swap case (only if 'tildeop' is set)
g~	g~	swap case
gu	gu	make lowercase
gU	gU	make uppercase
!	!	filter through an external program
=	=	filter through 'equalprg' or C-indenting if empty
gq	gq	text formatting
gw	gw	text formatting with no cursor movement
g?	g?	ROT13 encoding
>	>	shift right
<	<	shift left
zf	zf	define a fold
g@	g@	call function set with the 'operatorfunc' option

motion-count-multiplied

If the motion includes a count and the operator also had a count before it, the two counts are multiplied. For example: "2d3w" deletes six words.

operator-doubled

When doubling the operator it operates on a line. When using a count, before or after the first character, that many lines are operated upon. Thus 3dd deletes three lines. A count before and after the first character is multiplied, thus 2y3y yanks six lines.

operator-resulting-pos

After applying the operator the cursor is mostly left at the start of the text that was operated upon. For example, "yfe" doesn't move the cursor, but "yFe" moves the cursor leftwards to the "e" where the yank started.
The 'startofline' option applies only to the "d", "<<", "==" and ">>" linewise operations.

linewise charwise characterwise

The operator either affects whole lines, or the characters between the start and end position. Generally, motions that move between lines affect lines (are linewise), and motions that move within a line affect characters (are charwise). However, there are some exceptions.

exclusive inclusive

Character motion is either inclusive or exclusive. When inclusive, the start and end position of the motion are included in the operation. When exclusive, the last character towards the end of the buffer is not included. Linewise motions always include the start and end position. Plugins can check the v:event.inclusive flag of the TextYankPost event.

Which motions are linewise, inclusive or exclusive is mentioned with the command. There are however, two general exceptions:

1. If the motion is exclusive and the end of the motion is in column 1, the end of the motion is moved to the end of the previous line and the motion becomes inclusive. Example: "}" moves to the first line after a paragraph, but "d}" will not include that line.

   exclusive-linewise

2. If the motion is exclusive, the end of the motion is in column 1 and the start of the motion was at or before the first non-blank in the line, the motion becomes linewise. Example: If a paragraph begins with some blanks and you do "d}" while standing on the first non-blank, all the lines of the paragraph are deleted, including the blanks. If you do a put now, the deleted lines will be inserted below the cursor position.

Note that when the operator is pending (the operator command is typed, but the motion isn't yet), a special set of mappings can be used. See :omap.

Instead of first giving the operator and then a motion you can use Visual mode: mark the start of the text with "v", move the cursor to the end of the text that is to be affected and then hit the operator. The text between the start and the cursor position is highlighted, so you can see what text will be operated upon. This allows much more freedom, but requires more key strokes and has limited redo functionality. See the chapter on Visual mode Visual-mode.

You can use a ":" command for a motion. For example "d:call FindEnd()". But this can't be repeated with "." if the command is more than one line. This can be repeated:

```
d:call search("f")<CR>
```

This cannot be repeated:

```
d:if 1<CR>
   call search("f")<CR>
endif<CR>
```

Note that when using ":" any motion becomes charwise exclusive.

inclusive-motion-selection-exclusive

When 'selection' is "exclusive", Visual mode is active and an inclusive motion has been used, the cursor position will be adjusted by another character to the right, so that the Visual selection includes the expected text and can be acted upon.

forced-motion

FORCING A MOTION TO BE LINEWISE, CHARWISE OR BLOCKWISE

When a motion is not of the type you would like to use, you can force another type by using "v", "V" or CTRL-V just after the operator.
Example:

```
dj
```

deletes two lines

```
dvj
```

deletes from the cursor position until the character below the cursor

```
d<C-V>j
```

deletes the character under the cursor and the character below the cursor.

Be careful with forcing a linewise movement to be used charwise or blockwise, the column may not always be defined.

o_v

v When used after an operator, before the motion command: Force the operator to work charwise, also when the motion is linewise. If the motion was linewise, it will become exclusive.
If the motion already was charwise, toggle inclusive/exclusive. This can be used to make an exclusive motion inclusive and an inclusive motion exclusive.

o_V

V When used after an operator, before the motion command: Force the operator to work linewise, also when the motion is charwise.

o_CTRL-V

CTRL-V When used after an operator, before the motion command: Force the operator to work blockwise. This works like Visual block mode selection, with the corners defined by the cursor position before and after the motion.

2. Left-right motions left-right-motions

These commands move the cursor to the specified column in the current line. They stop at the first column and at the end of the line, except "$", which may move to one of the next lines. See 'whichwrap' option to make some of the commands move across line boundaries.

h or h

<Left> or <Left>

CTRL-H or CTRL-H <BS>
<BS> [count] characters to the left. exclusive motion.
Note: If you prefer <BS> to delete a character, use
the mapping:

```
:map CTRL-V<BS>          X
```

(to enter "CTRL-V<BS>" type the CTRL-V key, followed
by the <BS> key)

l or l

<Right> or <Right> <Space>
<Space> [count] characters to the right. exclusive motion.
See the 'whichwrap' option for adjusting the behavior
at end of line

0

0 To the first character of the line. exclusive
motion.

<Home> <kHome>
<Home> To the first character of the line. exclusive
motion. When moving up or down next, stay in same
TEXT column (if possible). Most other commands stay
in the same SCREEN column. <Home> works like "1|",
which differs from "0" when the line starts with a
<Tab>.

^

^ To the first non-blank character of the line.
exclusive motion. Any count is ignored.

$ <End> <kEnd>
$ or <End> To the end of the line. When a count is given also go
[count - 1] lines downward, or as far is possible.
inclusive motion. If a count of 2 or larger is
given and the cursor is on the last line, that is an
error and the cursor doesn't move.
In Visual mode the cursor goes to just after the last
character in the line.
When 'virtualedit' is active, "$" may move the cursor
back from past the end of the line to the last
character in the line.

g_

g_ To the last non-blank character of the line and
[count - 1] lines downward inclusive.

g0 g<Home>
g0 or g<Home> When lines wrap ('wrap' on): To the first character of
the screen line. exclusive motion. Differs from
"0" when a line is wider than the screen.
When lines don't wrap ('wrap' off): To the leftmost
character of the current line that is on the screen.
Differs from "0" when the first character of the line
is not on the screen.

g^

g^ When lines wrap ('wrap' on): To the first non-blank
character of the screen line. exclusive motion.
Differs from "^" when a line is wider than the screen.
When lines don't wrap ('wrap' off): To the leftmost
non-blank character of the current line that is on the
screen. Differs from "^" when the first non-blank
character of the line is not on the screen.

gm

gm Like "g0", but half a screenwidth to the right (or as
much as possible).

gM

gM Like "g0", but to halfway the text of the line.
With a count: to this percentage of text in the line.
Thus "10gM" is near the start of the text and "90gM"
is near the end of the text.

g$

g$ When lines wrap ('wrap' on): To the last character of
the screen line and [count - 1] screen lines downward
inclusive. Differs from "$" when a line is wider
than the screen.
When lines don't wrap ('wrap' off): To the rightmost
character of the current line that is visible on the
screen. Differs from "$" when the last character of
the line is not on the screen or when a count is used.
Additionally, vertical movements keep the column,
instead of going to the end of the line.

When 'virtualedit' is enabled moves to the end of the screen line.

g<End> g<kEnd>

g<End> Like g$ but to the last non-blank character instead of the last character.

bar

| To screen column [count] in the current line. exclusive motion. Ceci n'est pas une pipe.

f

f{char} To [count]'th occurrence of {char} to the right. The cursor is placed on {char} inclusive.
{char} can be entered as a digraph digraph-arg.
Composing characters may be used utf-8-char-arg.
:lmap mappings apply to {char}. The CTRL-^ command in Insert mode can be used to switch this on/off i_CTRL-^.

See getcharsearch() setcharsearch()

F

F{char} To the [count]'th occurrence of {char} to the left. The cursor is placed on {char} exclusive.
{char} can be entered like with the f command.

t

t{char} Till before [count]'th occurrence of {char} to the right. The cursor is placed on the character left of {char} inclusive.
{char} can be entered like with the f command.

See getcharsearch() setcharsearch()

T

T{char} Till after [count]'th occurrence of {char} to the left. The cursor is placed on the character right of {char} exclusive.
{char} can be entered like with the f command.

;

; Repeat latest f, t, F or T [count] times.

See getcharsearch() setcharsearch() cpo-;

,

, Repeat latest f, t, F or T in opposite direction [count] times.

See getcharsearch() setcharsearch() cpo-;

3. Up-down motions up-down-motions

k or k

<Up> or <Up> CTRL-P
CTRL-P [count] lines upward linewise.

j or j

<Down> or <Down>

CTRL-J or CTRL-J

<NL> or <NL> CTRL-N
CTRL-N [count] lines downward linewise.

gk or gk g<Up>
g<Up> [count] display lines upward. exclusive motion.
Differs from 'k' when lines wrap, and when used with an operator, because it's not linewise.

gj or gj g<Down>
g<Down> [count] display lines downward. exclusive motion.
Differs from 'j' when lines wrap, and when used with an operator, because it's not linewise.

-

- <minus> [count] lines upward, on the first non-blank character linewise.

-_default

By default "-" is mapped to open the parent directory of the current buffer. dir-mappings default-mappings

+ or +

CTRL-M or CTRL-M <CR>
<CR> [count] lines downward, on the first non-blank character linewise.

-

_ <underscore>		[count] - 1 lines downward, on the first non-blank
			character linewise.

G

G			Goto line [count], default last line, on the first
			non-blank character linewise. If 'startofline' not
			set, keep the same column.
			G is one of the jump-motions.

<C-End>

<C-End>			Goto line [count], default last line, on the last
			character inclusive.

<C-Home>	or			gg <C-Home>
gg			Goto line [count], default first line, on the first
			non-blank character linewise. If 'startofline' not
			set, keep the same column.

:[range]

:[range]		Set the cursor on the last line number in [range].
			In Ex mode, print the lines in [range].
			[range] can also be just one line number, e.g., ":1"
			or ":'m".
			In contrast with G this command does not modify the
			jumplist.

N%

{count}%		Go to {count} percentage in the file, on the first
			non-blank in the line linewise. To compute the new
			line number this formula is used:
			    ({count} * number-of-lines + 99) / 100
			See also 'startofline' option.

:[range]go[to] [count]			:go :goto go
[count]go		Go to [count] byte in the buffer. exclusive motion.
			Default [count] is one, start of the file. When
			giving [range], the last number in it used as the byte
			count. End-of-line characters are counted depending
			on the current 'fileformat' setting.
			Also see the line2byte() function, and the 'o'
			option in 'statusline'.

These commands move to the specified line. They stop when reaching the first
or the last line. The first two commands put the cursor in the same column
(if possible) as it was after the last command that changed the column,
except after the "$" command, then the cursor will be put on the last
character of the line.

4. Word motions

word-motions

<S-Right>	or			<S-Right> w
w			[count] words forward. exclusive motion.

<C-Right>	or			<C-Right> W
W			[count] WORDS forward. exclusive motion.

e

e			Forward to the end of word [count] inclusive.
			Does not stop in an empty line.

E

E			Forward to the end of WORD [count] inclusive.
			Does not stop in an empty line.

<S-Left>	or			<S-Left> b
b			[count] words backward. exclusive motion.

<C-Left>	or			<C-Left> B
B			[count] WORDS backward. exclusive motion.

ge

ge			Backward to the end of word [count] inclusive.

gE

gE			Backward to the end of WORD [count] inclusive.

These commands move over words or WORDS.

word

A word consists of a sequence of letters, digits and underscores, or a
sequence of other non-blank characters, separated with white space (spaces,
tabs, <EOL>). This can be changed with the 'iskeyword' option. For
characters above 255, a word ends when the Unicode character class changes
(e.g., between letters, subscripts, emojis, etc). An empty line is also
considered to be a word.

WORD

A WORD consists of a sequence of non-blank characters, separated with white space. An empty line is also considered to be a WORD.

A sequence of folded lines is counted for one word of a single character. "w" and "W", "e" and "E" move to the start/end of the first word or WORD after a range of folded lines. "b" and "B" move to the start of the first word or WORD before the fold.

Special case: "cw" and "cW" are treated like "ce" and "cE" if the cursor is on a non-blank. This is Vi-compatible, see cpo- to change the behavior.

Another special case: When using the "w" motion in combination with an operator and the last word moved over is at the end of a line, the end of that word becomes the end of the operated text, not the first word in the next line.

The original Vi implementation of "e" is buggy. For example, the "e" command will stop on the first character of a line if the previous line was empty. But when you use "2e" this does not happen. In Vim "ee" and "2e" are the same, which is more logical. However, this causes a small incompatibility between Vi and Vim.

5. Text object motions object-motions

(

([count] sentences backward. exclusive motion.

)

) [count] sentences forward. exclusive motion.

{

{ [count] paragraphs backward. exclusive motion.

}

} [count] paragraphs forward. exclusive motion.

]]

]] [count] sections forward or to the next "{" in the first column. When used after an operator, then also stops below a "}" in the first column. exclusive
Note that exclusive-linewise often applies.
In a :terminal buffer each shell prompt is treated as a section. terminal_]]

][

][[count] sections forward or to the next '}' in the first column. exclusive
Note that exclusive-linewise often applies.

[[

[[[count] sections backward or to the previous "{" in the first column. exclusive
Note that exclusive-linewise often applies.
In a :terminal buffer each shell prompt is treated as a section. terminal_]]

[]

[] [count] sections backward or to the previous "}" in the first column. exclusive
Note that exclusive-linewise often applies.

These commands move over three kinds of text objects.

sentence

A sentence is defined as ending at a '.', '!' or '?' followed by either the end of a line, or by a space or tab. Any number of closing ')', ']', '"' and ''' characters may appear after the '.', '!' or '?' before the spaces, tabs or end of line. A paragraph and section boundary is also a sentence boundary.
If the 'J' flag is present in 'cpoptions', at least two spaces have to follow the punctuation mark; <Tab>s are not recognized as white space.
The definition of a sentence cannot be changed.

paragraph

A paragraph begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the 'paragraphs' option. The default is "IPLPPPQPP TPHPLIPpLpItpplpipbp", which corresponds to the macros ".IP", ".LP", etc. (These are nroff macros, so the dot must be in the first column). A section boundary is also a paragraph boundary.
Note that a blank line (only containing white space) is NOT a paragraph boundary.
Note: this does not include a '{' or '}' in the first column.

section

A section begins after a form-feed (<C-L>) in the first column and at each of a set of section macros, specified by the pairs of characters in the 'sections' option. The default is "SHNHH HUnhsh", which defines a section to

start at the nroff macros ".SH", ".NH", ".H", ".HU", ".nh" and ".sh".
In a :terminal buffer each shell prompt is treated as a section. terminal_]]

The "]]" and "[[" commands stop at the '{' in the first column. This is useful to find the start of a function in a C program. To search for a '}' in the first column, the end of a C function, use "][" (forward) or "[]" (backward). Note that the first character of the command determines the search direction.

If your '{' or '}' are not in the first column, and you would like to use "[[" and "]]" anyway, try these mappings:

```
:map [[ ?{<CR>w99[{
:map ][ /}<CR>b99]}
:map ]] j0[[%/{<CR>
:map [] k$][%?}<CR>
```

[type these literally, see <>]

6. Text object selection

object-select text-objects

v_a v_i

This is a series of commands that can only be used while in Visual mode or after an operator. The commands that start with "a" select "a"n object including white space, the commands starting with "i" select an "inner" object without white space, or just the white space. Thus the "inner" commands always select less text than the "a" commands.

Also see gn and gN, operating on the last search pattern.

v_aw aw

aw "a word", select [count] words (see word).
Leading or trailing white space is included, but not counted.
When used in Visual linewise mode "aw" switches to Visual charwise mode.

v_iw iw

iw "inner word", select [count] words (see word).
White space between words is counted too.
When used in Visual linewise mode "iw" switches to Visual charwise mode.

v_aW aW

aW "a WORD", select [count] WORDs (see WORD).
Leading or trailing white space is included, but not counted.
When used in Visual linewise mode "aW" switches to Visual charwise mode.

v_iW iW

iW "inner WORD", select [count] WORDs (see WORD).
White space between words is counted too.
When used in Visual linewise mode "iW" switches to Visual charwise mode.

v_as as

as "a sentence", select [count] sentences (see sentence).
When used in Visual mode it is made charwise.

v_is is

is "inner sentence", select [count] sentences (see sentence).
When used in Visual mode it is made charwise.

v_ap ap

ap "a paragraph", select [count] paragraphs (see paragraph).
Exception: a blank line (only containing white space) is also a paragraph boundary.
When used in Visual mode it is made linewise.

v_ip ip

ip "inner paragraph", select [count] paragraphs (see paragraph).
Exception: a blank line (only containing white space) is also a paragraph boundary.
When used in Visual mode it is made linewise.

v_al al

al "all lines", select the whole buffer.
When used in Visual mode it is made linewise.

v_il il

il "inner line", select the current line without leading or trailing white space. Fails on a blank or white-space-only line.
When used in Visual mode it selects the line under the cursor and switches to charwise Visual mode.

v_a] v_a[a] a[

a]
a["a [] block", select [count] '[' ']' blocks. This goes backwards to the [count] unclosed '[', and finds the matching ']'. The enclosed text is selected, including the '[' and ']'. The cpo-M option flag

is used to handle escaped brackets.
When used in Visual mode it is made charwise.

i] v_i] v_i[i] i[
i["inner [] block", select [count] '[' ']' blocks. This goes backwards to the [count] unclosed '[', and finds the matching ']'. The enclosed text is selected, excluding the '[' and ']'. It's an error to select an empty inner block like "[]". The cpo-M option flag is used to handle escaped brackets.
When used in Visual mode it is made charwise.

a) v_a) a) a(
a(vab v_ab v_a(ab
ab "a block", select [count] blocks, from "[count] [(" to the matching ')', including the '(' and ')' (see [(). Does not include white space outside of the parenthesis. The cpo-M option flag is used to handle escaped parenthesis.
When used in Visual mode it is made charwise.

i) v_i) i) i(
i(vib v_ib v_i(ib
ib "inner block", select [count] blocks, from "[count] [(" to the matching ')', excluding the '(' and ')' (see [(). If the cursor is not inside a () block, then find the next "(". It's an error to select an empty inner block like "()". The cpo-M option flag is used to handle escaped parenthesis.
When used in Visual mode it is made charwise.

a> v_a> v_a< a> a<
a< "a <> block", select [count] <> blocks, from the [count]'th unmatched '<' backwards to the matching '>', including the '<' and '>'. The cpo-M option flag is used to handle escaped '<' and '>'.
When used in Visual mode it is made charwise.

i> v_i> v_i< i> i<
i< "inner <> block", select [count] <> blocks, from the [count]'th unmatched '<' backwards to the matching '>', excluding the '<' and '>'. It's an error to select an empty inner block like "<>". The cpo-M option flag is used to handle escaped '<' and '>'.
When used in Visual mode it is made charwise.

v_at at
at "a tag block", select [count] tag blocks, from the [count]'th unmatched "<aaa>" backwards to the matching "</aaa>", including the "<aaa>" and "</aaa>".
See tag-blocks about the details.
When used in Visual mode it is made charwise.

v_it it
it "inner tag block", select [count] tag blocks, from the [count]'th unmatched "<aaa>" backwards to the matching "</aaa>", excluding the "<aaa>" and "</aaa>".
See tag-blocks about the details.
When used in Visual mode it is made charwise.

a} v_a} a} a{
a{ v_aB v_a{ aB
aB "a Block", select [count] Blocks, from [count] [{ to the matching "}", including the "{" and "}" (see [{). The cpo-M option flag is used to handle escaped braces.
When used in Visual mode it is made charwise.

i} v_i} i} i{
i{ v_iB v_i{ iB
iB "inner Block", select [count] Blocks, from [count] [{ to the matching "}", excluding the "{" and "}" (see [{). It"s an error to select an empty inner block like "{}". The cpo-M option flag is used to handle escaped braces.
When used in Visual mode it is made charwise.

a" v_aquote aquote
a' v_a' a'
a` v_a` a`
"a quoted string". Selects the text from the previous quote until the next quote. The 'quoteescape' option is used to skip escaped quotes.
Only works within one line.
When the cursor starts on a quote, Vim will figure out which quote pairs form a string by searching from the start of the line.
Any trailing white space is included, unless there is none, then leading white space is included.
When used in Visual mode it is made charwise.
Repeating this object in Visual mode another string is included. A count is currently not used.

i" v_iquote iquote
i' v_i' i'
i` v_i` i`

Like a", a' and a`, but exclude the quotes and
repeating won't extend the Visual selection.
Special case: With a count of 2 the quotes are
included, but no extra white space as with a"/a'/a`.

o_object-select

When used after an operator:
For non-block objects:
For the "a" commands: The operator applies to the object and the white space after the object. If there is no white space after the object or when the cursor was in the white space before the object, the white space before the object is included.
For the "inner" commands: If the cursor was on the object, the operator applies to the object. If the cursor was on white space, the operator applies to the white space.
For a block object:
The operator applies to the block where the cursor is in, or the block on which the cursor is on one of the braces. For the "inner" commands the surrounding braces are excluded. For the "a" commands, the braces are included.

v_object-select

When used in Visual mode:
When start and end of the Visual area are the same (just after typing "v"):
One object is selected, the same as for using an operator.
When start and end of the Visual area are not the same:
For non-block objects the area is extended by one object or the white space up to the next object, or both for the "a" objects. The direction in which this happens depends on which side of the Visual area the cursor is. For the block objects the block is extended one level outwards.

For illustration, here is a list of delete commands, grouped from small to big objects. Note that for a single character and a whole line the existing vi movement commands are used.

"dl"	delete character (alias: "x")	dl
"diw"	delete inner word	diw
"daw"	delete a word	daw
"diW"	delete inner WORD (see WORD)	diW
"daW"	delete a WORD (see WORD)	daW
"dgn"	delete the next search pattern match	dgn
"dd"	delete one line	dd
"dis"	delete inner sentence	dis
"das"	delete a sentence	das
"dib"	delete inner '(' ')' block	dib
"dab"	delete a '(' ')' block	dab
"dip"	delete inner paragraph	dip
"dap"	delete a paragraph	dap
"diB"	delete inner '{' '}' block	diB
"daB"	delete a '{' '}' block	daB

Note the difference between using a movement command and an object. The movement command operates from here (cursor position) to where the movement takes us. When using an object the whole object is operated upon, no matter where on the object the cursor is. For example, compare "dw" and "daw": "dw" deletes from the cursor position to the start of the next word, "daw" deletes the word under the cursor and the space after or before it.

Tag blocks tag-blocks

For the "it" and "at" text objects an attempt is done to select blocks between matching tags for HTML and XML. But since these are not completely compatible there are a few restrictions.

The normal method is to select a <tag> until the matching </tag>. For "at" the tags are included, for "it" they are excluded. But when "it" is repeated the tags will be included (otherwise nothing would change). Also, "it" used on a tag block with no contents will select the leading tag.

"<aaa/>" items are skipped. Case is ignored, also for XML where case does matter.

In HTML it is possible to have a tag like
 or <meta ...> without a matching end tag. These are ignored.

The text objects are tolerant about mistakes. Stray end tags are ignored.

Custom text object text-object-define

The following code example shows how to define your own custom text-object. Notice how the Visual-mode (xnoremap) mapping is defined as an <expr> mapping which sets the '['] marks, then the Operator-pending (onoremap) mapping is implemented using Visual-mode.

```
" text-object: entire buffer
function! s:line_outer_movement(count) abort
  if empty(getline(1)) && 1 == line('$')
    return "\<Esc>"
  endif
  let [lopen, copen, lclose, cclose] = [1, 1, line('$'), 1]
  call setpos("'[", [0, lopen, copen, 0])
  call setpos("']", [0, lclose, cclose, 0])
  return "'[o']"
endfunction
xnoremap <expr>   al <SID>line_outer_movement(v:count1)
onoremap <silent> al :normal Val<CR>

" text-object: line
function! s:line_inner_movement(count) abort
  "TODO: handle count
  if empty(getline('.'))
    return "\<Esc>"
  endif
  let [lopen, copen, lclose, cclose] = [line('.'), 1, line('.'), col('$')-1]
  call setpos("'[", [0, lopen, copen, 0])
  call setpos("']", [0, lclose, cclose, 0])
  return "`[o`]"
endfunction
xnoremap <expr>   il <SID>line_inner_movement(v:count1)
onoremap <silent> il :normal vil<CR>
```

7. Marks

mark-motions E20 E78

Jumping to a mark can be done in two ways:
1. With ` (backtick): The cursor is positioned at the specified location and the motion is exclusive.
2. With ' (single quote): The cursor is positioned on the first non-blank character in the line of the specified location and the motion is linewise.

mark-view

3. Apart from the above if 'jumpoptions' contains "view", they will also try to restore the mark view. This is the number of lines between the cursor position and the window topline (first buffer line displayed in the window) when it was set.

m m mark Mark

m{a-zA-Z} Set mark {a-zA-Z} at cursor position (does not move the cursor, this is not a motion command).

m' m`

m' or m` Set the previous context mark. This can be jumped to with the "''" or "``" command (does not move the cursor, this is not a motion command).

m[m]

m[or m] Set the '[or '] mark. Useful when an operator is to be simulated by multiple commands. (does not move the cursor, this is not a motion command).

m< m>

m< or m> Set the '< or '> mark. Useful to change what the gv command selects. (does not move the cursor, this is not a motion command).
Note that the Visual mode cannot be set, only the start and end position.

:ma :mark E191

:[range]ma[rk] {a-zA-Z'} Set mark {a-zA-Z'} at last line number in [range], column 0. Default is cursor line.

:k

:[range]k{a-zA-Z'} Same as :mark, but the space before the mark name can be omitted.

' 'a ` `a

'{a-z} `{a-z} Jump to the mark {a-z} in the current buffer.

'A '0 `A `0

'{A-Z0-9} `{A-Z0-9} To the mark {A-Z0-9} in the file where it was set (not a motion command when in another file).

g' g'a g` g`a

g'{mark} g`{mark} Jump to the {mark}, but don't change the jumplist when jumping within the current buffer. Example:

```
g`"
```

jumps to the last known position in a file.
See also :keepjumps.

:marks

:marks List all marks, and extmarks namespaces (use ":marks {ns}" to list them).

The '(, '), '{ and '} marks are not listed.
The first column has number zero.

E283

:marks {arg} List the marks that are mentioned in {arg} (not a motion command). For example:

```
:marks aB
```

to list marks 'a' and 'B'.
If {arg} is extmark namespace, the extmarks for current buffer are listed:

```
:marks nvim.multicursor
```

:delm :delmarks

:delm[arks] {marks} Delete the specified marks. Marks that can be deleted include A-Z and 0-9. You cannot delete the ' mark. They can be specified by giving the list of mark names, or with a range, separated with a dash. Spaces are ignored. Examples:

```
:delmarks a           deletes mark a
:delmarks a b 1    deletes marks a, b and 1
:delmarks Aa       deletes marks A and a
:delmarks p-z      deletes marks in the range p to z
:delmarks ^.[]     deletes marks ^ . [ ]
:delmarks \"          deletes mark "
```

:delm[arks]! Delete all marks for the current buffer, but not marks A-Z or 0-9. Also clear the changelist.

A mark is not visible in any way. It is just a position in the file that is remembered. Do not confuse marks with named registers, they are totally unrelated.

```
'a - 'z         lowercase marks, valid within one file
'A - 'Z         uppercase marks, also called file marks, valid between files
'0 - '9         numbered marks, set from shada file
```

Lowercase marks 'a to 'z are remembered as long as the file remains in the buffer list. If you remove the file from the buffer list, all its marks are lost. If you delete a line that contains a mark, that mark is erased.

Lowercase marks can be used in combination with operators. For example: "d't" deletes the lines from the cursor position to mark 't'. Hint: Use mark 't' for Top, 'b' for Bottom, etc.. Lowercase marks are restored when using undo and redo.

Uppercase marks 'A to 'Z include the file name. You can use them to jump from file to file. You can only use an uppercase mark with an operator if the mark is in the current file. The line number of the mark remains correct, even if you insert/delete lines or edit another file for a moment. When the 'shada' option is not empty, uppercase marks are kept in the .shada file. See shada-file-marks.

Numbered marks '0 to '9 are quite different. They can not be set directly. They are only present when using a shada file shada-file. Basically '0 is the location of the cursor when you last exited Vim, '1 the last but one time, etc. Use the "r" flag in 'shada' to specify files for which no Numbered mark should be stored. See shada-file-marks.

'[`[

'[`[To the first character of the previously changed, or yanked text. Also set when writing the buffer.

'] `]

'] `] To the last character of the previously changed or yanked text. Also set when writing the buffer.

After executing an operator the Cursor is put at the beginning of the text that was operated upon. After a put command ("p" or "P") the cursor is sometimes placed at the first inserted line and sometimes on the last inserted character. The four commands above put the cursor at either end. Example: After yanking 10 lines you want to go to the last one of them: "10Y']". After inserting several lines with the "p" command you want to jump to the lowest inserted line: "p']". This also works for text that has been inserted.

Note: After deleting text, the start and end positions are the same, except when using blockwise Visual mode. These commands do not work when no change was made yet in the current file.

'< `<

'< `< To the first line or character of the last selected Visual area in the current buffer. For block mode it may also be the last character in the first line (to be able to define the block).

'> `>

'> `> To the last line or character of the last selected

Visual area in the current buffer. For block mode it may also be the first character of the last line (to be able to define the block). Note that 'selection' applies, the position may be just after the Visual area.

'' ``

'' ``	To the position before the latest jump, or where the last "m'" or "m`" command was given. Not set when the :keepjumps command modifier was used.
Also see restore-position.

'quote `quote

'" `"	To the cursor position when last exiting the current buffer. Defaults to the first character of the first line. See last-position-jump for how to use this for each opened file.
Only one position is remembered per buffer, not one for each window. As long as the buffer is visible in a window the position won't be changed. Mark is also reset when :wshada is run.

'^ `^

'^ `^	To the position where the cursor was the last time when Insert mode was stopped. This is used by the gi command. Not set when the :keepjumps command modifier was used.

'. `.

'. `.	To the position where the last change was made. The position is at or near where the change started. Sometimes a command is executed as several changes, then the position can be near the end of what the command changed. For example when inserting a word, the position will be on the last character.
To jump to older changes use g;.

': `:

':	In a prompt buffer, the start of the current user input. Text from this mark until end of buffer will be submitted when the user submits the prompt.

'(`(

'(`(	To the start of the current sentence, like the |(| command.

') `)

') `)	To the end of the current sentence, like the |)| command.

'{ `{

'{ `{	To the start of the current paragraph, like the |{| command.

'} `}

'} `}	To the end of the current paragraph, like the |}| command.

These commands are not marks themselves, but jump to a mark:

]'

]'	[count] times to next line with a lowercase mark below the cursor, on the first non-blank character in the line.

]`

]`	[count] times to lowercase mark after the cursor.

['

['	[count] times to previous line with a lowercase mark before the cursor, on the first non-blank character in the line.

[`

[`	[count] times to lowercase mark before the cursor.

:loc[kmarks] {command}	:loc :lock :lockmarks
Execute {command} without adjusting marks. This is useful when changing text in a way that the line count will be the same when the change has completed.
WARNING: When the line count does change, marks below the change will keep their line number, thus move to another text line.
These items will not be adjusted for deleted/inserted lines:

- lower case letter marks 'a - 'z
- upper case letter marks 'A - 'Z
- numbered marks '0 - '9

- last insert position '^
- last change position '.
- last affected text area '[and ']
- the Visual area '< and '

```
- line numbers in placed signs
- line numbers in quickfix positions
- positions in the |jumplist|
- positions in the |tagstack|
These items will still be adjusted:
- previous context mark ''
- the cursor position
- the view of a window on a buffer
- folds
- diffs
```

:kee[pmarks] {command} :kee :keep :keepmarks
Currently only has effect for the filter command :range!:
- When the number of lines after filtering is equal to or larger than before, all marks are kept at the same line number.
- When the number of lines decreases, the marks in the lines that disappeared are deleted.

In any case the marks below the filtered text have their line numbers adjusted, thus stick to the text, as usual.
When the 'R' flag is missing from 'cpoptions' this has the same effect as using ":keepmarks".

:keepj :keepjumps
:keepj[umps] {command}
Moving around in {command} does not change the '', '. and '^ marks, the jumplist or the changelist.
Useful when making a change or inserting text automatically and the user doesn't want to go to this position. E.g., when updating a "Last change" timestamp in the first line:

```
:let lnum = line(".")
:keepjumps normal gg
:call SetLastChange()
:keepjumps exe "normal " .. lnum .. "G"
```

Note that ":keepjumps" must be used for every command. When invoking a function the commands in that function can still change the jumplist. Also, for :keepjumps exe 'command ' the "command" won't keep jumps. Instead use: :exe 'keepjumps command'

8. Jumps jump-motions

A "jump" is a command that normally moves the cursor several lines away. If you make the cursor "jump" the position of the cursor before the jump is remembered. You can return to that position with the "''" and "``" commands, unless the line containing that position was changed or deleted. The following commands are "jump" commands: "'", "`", "G", "/", "?", "n", "N", "%", "(", ")", "[[", "]]", "{", "}", ":s", ":tag", "L", "M", "H" and the commands that start editing a new file.

CTRL-O

CTRL-O Go to [count] Older cursor position in jump list (not a motion command).

<Tab> or CTRL-I <Tab>
CTRL-I Go to [count] newer cursor position in jump list (not a motion command).

NOTE: In the GUI and in a terminal supporting tui-modifyOtherKeys or tui-csiu, CTRL-I can be mapped separately from <Tab>, on the condition that both keys are mapped, otherwise the mapping applies to both. Except in tmux: https://github.com/tmux/tmux/issues/2705

:ju :jumps
:ju[mps] Print the jump list (not a motion command).

:cle :clearjumps
:cle[arjumps] Clear the jump list of the current window.

Jumps are remembered in a jump list. With the CTRL-O and CTRL-I command you can go to cursor positions before older jumps, and back again. Thus you can move up and down the list. There is a separate jump list for each window. The maximum number of entries is fixed at 100.

For example, after three jump commands you have this jump list:

```
 jump line  col file/text
   3     1    0 some text
   2    70    0 another line
   1  1154   23 end.
>
```

The "file/text" column shows the file name, or the text at the jump if it is in the current file (an indent is removed and a long line is truncated to fit in the window).

The marker ">" indicates the current position in the jumplist. It may not be shown when filtering the :jumps command using :filter

You are currently in line 1167. If you then use the CTRL-O command, the cursor is put in line 1154. This results in:

```
 jump line  col file/text
   2     1    0 some text
   1    70    0 another line
>  0  1154   23 end.
   1  1167    0 foo bar
```

The pointer will be set at the last used jump position. The next CTRL-O command will use the entry above it, the next CTRL-I command will use the entry below it. If the pointer is below the last entry, this indicates that you did not use a CTRL-I or CTRL-O before. In this case the CTRL-O command will cause the cursor position to be added to the jump list, so you can get back to the position before the CTRL-O. In this case this is line 1167.

With more CTRL-O commands you will go to lines 70 and 1. If you use CTRL-I you can go back to 1154 and 1167 again. Note that the number in the "jump" column indicates the count for the CTRL-O or CTRL-I command that takes you to this position.

If you use a jump command, the current line number is inserted at the end of the jump list. If the same line was already in the jump list, it is removed. The result is that when repeating CTRL-O you will get back to old positions only once.

When the :keepjumps command modifier is used, jumps are not stored in the jumplist. Jumps are also not stored in other cases, e.g., in a :global command. You can explicitly add a jump by setting the ' mark with "m'". Note that calling setpos() does not do this.

After the CTRL-O command that got you into line 1154 you could give another jump command (e.g., "G"). The jump list would then become:

```
 jump line  col file/text
   4     1    0 some text
   3    70    0 another line
   2  1167    0 foo bar
   1  1154   23 end.
>
```

The line numbers will be adjusted for deleted and inserted lines. This fails if you stop editing a file without writing, like with ":n!".

When you split a window, the jumplist will be copied to the new window.

If you have included the ' item in the 'shada' option the jumplist will be stored in the ShaDa file and restored when starting Vim.

When 'jumpoptions' option includes "stack", the jumplist behaves like the tag stack. When jumping to a new location from the middle of the jumplist, the locations after the current position will be discarded. With this option set you can move through a tree of jump locations. When going back up a branch and then down another branch, CTRL-O still takes you further up the tree.

Given a jumplist like the following in which CTRL-O has been used to move back three times to location X:

```
 jump line  col file/text
   2  1260    8 mark.c          <-- location X-2
   1   685    0 eval.c          <-- location X-1
>  0   462   36 eval.c          <-- location X
   1   479   39 eval.c
   2   213    2 mark.c
   3   181    0 mark.c
```

jumping to (new) location Y results in the locations after the current locations being removed:

```
 jump line  col file/text
   3  1260    8 mark.c          <-- location X-2
   2   685    0 eval.c          <-- location X-1
   1   462   36 eval.c          <-- location X
>
```

Then, when yet another location Z is jumped to, the new location Y appears directly after location X in the jumplist and location X remains in the same position relative to the locations (X-1, X-2, etc., ...) that had been before it prior to the original jump from X to Y:

```
 jump line  col file/text
   4  1260    8 mark.c          <-- location X-2
   3   685    0 eval.c          <-- location X-1
   2   462   36 eval.c          <-- location X
   1   100    0 buffer.c               <-- location Y
>
```

CHANGE LIST JUMPS changelist change-list-jumps E664

When making a change the cursor position is remembered. One position is remembered for every change that can be undone, unless it is close to a previous change. Two commands can be used to jump to positions of changes, also those that have been undone:

g; E662

g; Go to [count] older position in change list.
If [count] is larger than the number of older change positions go to the oldest change.
If there is no older change an error message is given.
(not a motion command)

g, E663

g, Go to [count] newer position in change list.
Just like g; but in the opposite direction.
(not a motion command)

When using a count you jump as far back or forward as possible. Thus you can use "999g;" to go to the first change for which the position is still remembered. The number of entries in the change list is fixed and is the same as for the jumplist.

When two undo-able changes are in the same line and at a column position less than 'textwidth' apart only the last one is remembered. This avoids that a sequence of small changes in a line, for example "xxxxx", adds many positions to the change list. When 'textwidth' is zero 'wrapmargin' is used. When that also isn't set a fixed number of 79 is used. Detail: For the computations bytes are used, not characters, to avoid a speed penalty (this only matters for multibyte encodings).

Note that when text has been inserted or deleted the cursor position might be a bit different from the position of the change. Especially when lines have been deleted.

When the :keepjumps command modifier is used the position of a change is not remembered.

:changes

:changes Print the change list. A ">" character indicates the current position. Just after a change it is below the newest entry, indicating that g; takes you to the newest entry position. The first column indicates the count needed to take you to this position. Example:

change line col text

```
                                3     9    8 bla bla bla
                                2    11   57 foo is a bar
                                1    14   54 the latest changed line
```

```
The `3g;` command takes you to line 9.  Then the
output of `:changes` is:

        change line  col text ~
        >   0     9    8 bla bla bla
            1    11   57 foo is a bar
            2    14   54 the latest changed line

Now you can use "g," to go to line 11 and "2g," to go
to line 14.
```

9. Various motions various-motions

%

% Find the next item in this line after or under the cursor and jump to its match. inclusive motion.
Items can be:
([{}]) parenthesis or (curly/square) brackets
(this can be changed with the 'matchpairs' option)
/* */ start or end of C-style comment

#if, #ifdef, #else, #elif, #endif
C preprocessor conditionals (when the cursor is on the # or no ([{ is following)
For other items the matchit plugin can be used, see |matchit|. This plugin also helps to skip matches in comments.

When 'cpoptions' contains "M" cpo-M backslashes before parens and braces are ignored. Without "M" the number of backslashes matters: an even number doesn't match with an odd number. Thus in "(\))" and "\((\)" the first and last parenthesis match.

When the '%' character is not present in 'cpoptions' cpo-%, parens and braces inside double quotes are ignored, unless the number of parens/braces in a line is uneven and this line and the previous one does not end in a backslash. '(', '{', '[', ']', '}' and ')' are also ignored (parens and braces inside single quotes). Note that this works fine for C, but not for Perl, where single quotes are used for strings.

When in addition 'comments' defines C-style "//" or "/*" comments, parens and braces inside such comments are skipped, like the = operator does. This does not happen when the cursor is inside a comment, so a match inside that comment can still be found. Otherwise you can use the |matchit| plugin or put quotes around matches.

No count is allowed, {count}% jumps to a line {count} percentage down the file N%. Using '%' on #if/#else/#endif makes the movement linewise.

[(

[(Go to [count] previous unmatched '('. exclusive motion.

[{

[{ Go to [count] previous unmatched '{'. exclusive motion.

])

]) Go to [count] next unmatched ')'. exclusive motion.

]}

]} Go to [count] next unmatched '}'. exclusive motion.

The above four commands can be used to go to the start or end of the current code block. It is like doing "%" on the "(", ")", "{" or "}" at the other end of the code block, but you can do this from anywhere in the code block. Very useful for C programs. Example: When standing on "case x:", [{ will bring you back to the switch statement.

]m

]m Go to [count] next start of a method (for Java or similar structured language). When not before the start of a method, jump to the start or end of the class. exclusive motion.

]M

]M Go to [count] next end of a method (for Java or similar structured language). When not before the end of a method, jump to the start or end of the class. exclusive motion.

[m

[m Go to [count] previous start of a method (for Java or similar structured language). When not after the start of a method, jump to the start or end of the class. When no '{' is found before the cursor this is an error. exclusive motion.

[M

[M Go to [count] previous end of a method (for Java or similar structured language). When not after the end of a method, jump to the start or end of the class. When no '}' is found before the cursor this is an error. exclusive motion.

The above four commands assume that the file contains a class with methods. The class definition is surrounded in '{' and '}'. Each method in the class is also surrounded with '{' and '}'. This applies to the Java language. The file looks like this:

```
// comment
class foo {
        int method_one() {
                body_one();
        }
        int method_two() {
                body_two();
        }
}
```

[To try this out copy the text and put it in a new buffer, the help text above confuses the jump commands]

Starting with the cursor on "body_two()", using "[m" will jump to the '{' at the start of "method_two()" (obviously this is much more useful when the method is long!). Using "2[m" will jump to the start of "method_one()". Using "3[m" will jump to the start of the class.

[#

[# Go to [count] previous unmatched "#if" or "#else". exclusive motion.

]#

]# Go to [count] next unmatched "#else" or "#endif". exclusive motion.

These two commands work in C programs that contain #if/#else/#endif constructs. It brings you to the start or end of the #if/#else/#endif where the current line is included. You can then use "%" to go to the matching line.

[star [/

[* or [/ Go to [count] previous start of a C comment "/*". exclusive motion.

]star]/

]* or]/ Go to [count] next end of a C comment "*/". exclusive motion.

H

H To line [count] from top (Home) of window (default: first line on the window) on the first non-blank character linewise. See also 'startofline' option. Cursor is adjusted for 'scrolloff' option, unless an operator is pending, in which case the text may scroll. E.g. "yH" yanks from the first visible line until the cursor line (inclusive).

M

M To Middle line of window, on the first non-blank character linewise. See also 'startofline' option.

L

L To line [count] from bottom of window (default: Last line on the window) on the first non-blank character linewise. See also 'startofline' option. Cursor is adjusted for 'scrolloff' option, unless an operator is pending, in which case the text may scroll. E.g. "yL" yanks from the cursor to the last visible line.

<LeftMouse> Moves to the position on the screen where the mouse click is exclusive. See also <LeftMouse>. If the position is in a status line, that window is made the active window and the cursor is not moved.

Insert-mode

Inserting and replacing text mode-ins-repl

Most of this file is about Insert and Replace mode. At the end are a few commands for inserting text in other ways.

An overview of the most often used commands can be found in chapter 24 of the user manual usr_24.txt.

Also see 'virtualedit', for moving the cursor to positions where there is no character. Useful for editing a table.

1. Special keys ins-special-keys

In Insert and Replace mode, the following characters have a special meaning; other characters are inserted directly. To insert one of these special characters into the buffer, precede it with CTRL-V. To insert a <Nul> character use "CTRL-V CTRL-@" or "CTRL-V 000". On some systems, you have to use "CTRL-V 003" to insert a CTRL-C. **Note:** When CTRL-V is mapped you can often use CTRL-Q instead i_CTRL-Q.

If you are working in a special language mode when inserting text, see the 'langmap' option, 'langmap', on how to avoid switching this mode on and off all the time.

char	action
<Esc> or CTRL-[	End insert or Replace mode, go back to Normal mode. Finish abbreviation. **Note:** If your <Esc> key is hard to hit, try CTRL-[instead. i_META i_ALT ALT (META) may act like <Esc> if the chord is not mapped. For example <A-x> acts like <Esc>x if <A-x> does not have an insert-mode mapping. i_CTRL-C
CTRL-C	Quit insert mode, go back to Normal mode. Do not check for abbreviations. Does not trigger the InsertLeave autocommand event.

i_CTRL-@

CTRL-@ Insert previously inserted text and stop insert.

i_CTRL-A

CTRL-A Insert previously inserted text.

i_CTRL-H i_<BS> i_BS

<BS> or CTRL-H Delete the character before the cursor (see i_backspacing about joining lines).

i_<Del> i_DEL

<Del> Delete the character under the cursor. If the cursor is at the end of the line, and the 'backspace' option includes "eol" (the default), delete the <EOL>; the next line is appended after the current one.

i_CTRL-W

CTRL-W Delete the word before the cursor (see i_backspacing about joining lines). See the section "word motions", word-motions, for the definition of a word.

i_CTRL-W-default

By default, sets a new undo point before deleting.
default-mappings

i_CTRL-U

CTRL-U Delete all characters that were entered after starting Insert mode and before the cursor in the current line.
If there are no newly entered characters and 'backspace' is not empty, delete all characters before the cursor in the current line.
If C-indenting is enabled the indent will be adjusted if the line becomes blank.
See i_backspacing about joining lines.

i_CTRL-U-default

By default, sets a new undo point before deleting.
default-mappings

i_CTRL-I i_<Tab> i_Tab

<Tab> or CTRL-I Insert a tab. If the 'expandtab' option is on, the equivalent number of spaces is inserted (use CTRL-V <Tab> to avoid the expansion; use CTRL-Q <Tab> if CTRL-V is mapped i_CTRL-Q). See also the 'smarttab' option and ins-expandtab.

i_CTRL-J i_<NL>

<NL> or CTRL-J Begin new line.

i_CTRL-M i_<CR>

<CR> or CTRL-M Begin new line.

i_CTRL-K

CTRL-K {char1} [char2]
Enter digraph (see digraphs). When {char1} is a special key, the code for that key is inserted in <> form. For example, the string "<S-Space>" can be entered by typing <C-K><S-Space> (two keys). Neither char is considered for mapping.

CTRL-N Find next keyword (see i_CTRL-N).
CTRL-P Find previous keyword (see i_CTRL-P).

CTRL-R {register} i_CTRL-R

Insert the contents of a register. Between typing CTRL-R and
the second character, '"' will be displayed to indicate that
you are expected to enter the name of a register. When used
with named or clipboard registers (A-Z,a-z,0-9,+) text is
inserted literally like pasting with "p". For other registers,
the text is inserted as if you typed it, but mappings and
abbreviations are not used. If you have options like
'textwidth', 'formatoptions', or 'autoindent' set, this will
influence what will be inserted. This is different from what
happens with the "p" command and pasting with the mouse.
Special registers:

```
'"'     the unnamed register, containing the text of
        the last delete or yank
'%'     the current file name
'#'     the alternate file name
"*"     the clipboard contents (X11: primary selection)
'+'     the clipboard contents
'/'     the last search pattern
':'     the last command-line
'.'     the last inserted text
```

i_CTRL-R_-

```
'-'     the last small (less than a line) delete
        register.  This is repeatable using . since
        it remembers the register to put instead of
        the literal text to insert.
```

i_CTRL-R_=

```
'='     the expression register: you are prompted to
        enter an expression (see expression)
        Note that 0x80 (128 decimal) is used for
        special keys.  E.g., you can use this to move
        the cursor up:
                CTRL-R ="\<Up>"
        you can use this to insert a register as
        typed with CTRL-R =@reg.
        Use CTRL-R CTRL-R to insert text literally.
        When the result is a List the items are used
        as lines.  They can have line breaks inside
        too.
        When the result is a Float it's automatically
        converted to a String.
        When append() or setline() is invoked the undo
        sequence will be broken.
```

See registers about registers.

CTRL-R CTRL-R {register} i_CTRL-R_CTRL-R

Insert the contents of a register. Works like using a single
CTRL-R, but the text is inserted literally, not as if typed.
This differs when the register contains characters like <BS>.
Example, where register a contains "ab^Hc":

```
CTRL-R a                results in "ac".
CTRL-R CTRL-R a         results in "ab^Hc".
```

Options 'textwidth', 'formatoptions', etc. still apply. If
you also want to avoid these, use CTRL-R CTRL-O, see below.
The '.' register (last inserted text) is still inserted as
typed.
After this command, the '.' register contains the text from
the register as if it was inserted by typing it.

CTRL-R CTRL-O {register} i_CTRL-R_CTRL-O

Insert the contents of a register literally and don't
auto-indent. Does the same as pasting with the mouse
<MiddleMouse>. When the register is linewise this will
insert the text above the current line, like with P.
The '.' register (last inserted text) is still inserted as
typed.
After this command, the '.' register contains the command
typed and not the text. I.e., the literals "^R^O" and not the
text from the register.
Does not replace characters in Replace-mode!

CTRL-R CTRL-P {register} i_CTRL-R_CTRL-P

Insert the contents of a register literally and fix the
indent, like [<MiddleMouse>.
The '.' register (last inserted text) is still inserted as
typed.
After this command, the '.' register contains the command
typed and not the text. I.e., the literals "^R^P" and not the
text from the register.
Does not replace characters in Replace-mode!

i_CTRL-T

CTRL-T Insert one shiftwidth of indent at the start of the current

line. The indent is always rounded to a 'shiftwidth'.

i_CTRL-D

CTRL-D Delete one shiftwidth of indent at the start of the current line. The indent is always rounded to a 'shiftwidth'.

i_0_CTRL-D

0 CTRL-D Delete all indent in the current line.

i_^_CTRL-D

^ CTRL-D Delete all indent in the current line. The indent is restored in the next line. This is useful when inserting a label.

i_CTRL-V

CTRL-V Insert next non-digit literally. It's also possible to enter the decimal, octal or hexadecimal value of a character i_CTRL-V_digit.
The characters typed right after CTRL-V are not considered for mapping.
For special keys, the CTRL modifier may be included into the key to produce a control character. If there is no control character for the key then its key-notation is inserted.
Note: When CTRL-V is mapped (e.g., to paste text) you can often use CTRL-Q instead i_CTRL-Q.

i_CTRL-Q

CTRL-Q Same as CTRL-V.
Note: Some terminal connections may eat CTRL-Q, it doesn't work then. It does work in the GUI.

CTRL-SHIFT-V i_CTRL-SHIFT-V i_CTRL-SHIFT-Q
CTRL-SHIFT-Q Works just like CTRL-V, but do not try to include the CTRL modifier into the key.
Note: When CTRL-SHIFT-V is intercepted by your system (e.g., to paste text) you can often use CTRL-SHIFT-Q instead. However, in some terminals (e.g. GNOME Terminal), CTRL-SHIFT-Q quits the terminal without confirmation.

CTRL-X Enter CTRL-X mode. This is a sub-mode where commands can be given to complete words or scroll the window. See i_CTRL-X and ins-completion.

i_CTRL-E

CTRL-E Insert the character which is below the cursor.

i_CTRL-Y

CTRL-Y Insert the character which is above the cursor.
Note that for CTRL-E and CTRL-Y 'textwidth' is not used, to be able to copy characters from a long line.

i_CTRL-_

CTRL-_ Switch between insert direction, by toggling 'revins'.

CTRL-_ moves the cursor to the end of the typed text.

This command is only available when the 'allowrevins' option is set.
Please refer to rileft.txt for more information about right-to-left mode.

i_CTRL-^

CTRL-^ Toggle the use of typing language characters.
When language :lmap mappings are defined:

- If 'iminsert' is 1 (langmap mappings used) it becomes 0 (no langmap mappings used).
- If 'iminsert' has another value it becomes 1, thus langmap mappings are enabled.

When no language mappings are defined:

- If 'iminsert' is 2 (Input Method used) it becomes 0 (no Input Method used).
- If 'iminsert' has another value it becomes 2, thus the Input Method is enabled.

When set to 1, the value of the "b:keymap_name" variable, the 'keymap' option or "<lang>" appears in the status line.
The language mappings are normally used to type characters that are different from what the keyboard produces. The 'keymap' option can be used to install a whole number of them.

i_CTRL-]

CTRL-] Trigger abbreviation, without inserting a character.

i_<Insert>

<Insert> Toggle between Insert and Replace mode.

The effect of the <BS>, CTRL-W, and CTRL-U depend on the 'backspace' option
(unless 'revins' is set). This is a comma-separated list of items:

item	action
indent	allow backspacing over autoindent
eol	allow backspacing over line breaks (join lines)
start	allow backspacing over the start of insert; CTRL-W and CTRL-U stop once at the start of insert.
nostop	like start, except CTRL-W and CTRL-U do not stop at the start of insert.

When 'backspace' is empty, Vi compatible backspacing is used. You cannot
backspace over autoindent, before column 1 or before where insert started.

If the 'backspace' option does contain "eol" and the cursor is in column 1
when one of the three keys is used, the current line is joined with the
previous line. This effectively deletes the <EOL> in front of the cursor.

i_CTRL-V_digit

With CTRL-V the decimal, octal or hexadecimal value of a character can be
entered directly. This way you can enter any character, except a line break
(<NL>, value 10). There are five ways to enter the character value:

first char	mode	max nr of chars	max value	
(none)	decimal	3	255	
o or O	octal	3	377	(255)
x or X	hexadecimal	2	ff	(255)
u	hexadecimal	4	ffff	(65535)
U	hexadecimal	8	7fffffff (2147483647)	

Normally you would type the maximum number of characters. Thus to enter a
space (value 32) you would type <C-V>032. You can omit the leading zero, in
which case the character typed after the number must be a non-digit. This
happens for the other modes as well: As soon as you type a character that is
invalid for the mode, the value before it will be used and the "invalid"
character is dealt with in the normal way.

If you enter a value of 10, it will end up in the file as a 0. The 10 is a
<NL>, which is used internally to represent the <Nul> character. When writing
the buffer to a file, the <NL> character is translated into <Nul>. The <NL>
character is written at the end of each line. Thus if you want to insert a
<NL> character in a file you will have to make a line break.
Also see 'fileformat'.

i_CTRL-X insert_expand

CTRL-X enters a sub-mode where several commands can be used. Most of these
commands do keyword completion; see ins-completion.

Two commands can be used to scroll the window up or down, without exiting
insert mode:

i_CTRL-X_CTRL-E

CTRL-X CTRL-E scroll window one line up.
               When doing completion look here: complete_CTRL-E

i_CTRL-X_CTRL-Y

CTRL-X CTRL-Y scroll window one line down.
               When doing completion look here: complete_CTRL-Y

After CTRL-X is pressed, each CTRL-E (CTRL-Y) scrolls the window up (down) by
one line unless that would cause the cursor to move from its current position
in the file. As soon as another key is pressed, CTRL-X mode is exited and
that key is interpreted as in Insert mode.

2. Special special keys ins-special-special

The following keys are special. They stop the current insert, do something,
and then restart insertion. This means you can do something without getting
out of Insert mode. This is very handy if you prefer to use the Insert mode
all the time, just like editors that don't have a separate Normal mode. You
can use CTRL-O if you want to map a function key to a command.

ins-repeat

The changes (inserted or deleted characters) before and after these keys can
be undone separately. Relative cursor-moves (the cursor keys, <S-Left>,
<Home>, CTRL-G j, …) are part of the insert: . repeats the whole insert,
including cursor-moves. After a jump (mouse, scroll, <PageUp>, …) only the
last change can be redone, and it behaves like an "i" command. The quote.
register always holds just the text typed after the last cursor-move.

char	action	
<Up> cursor one line up **i_<Up>**		
<Down>	cursor one line down	i_<Down>
CTRL-G <Up>	cursor one line up, insert start column	i_CTRL-G_<Up>
CTRL-G k	cursor one line up, insert start column	i_CTRL-G_k
CTRL-G CTRL-K	cursor one line up, insert start column	i_CTRL-G_CTRL-K
CTRL-G <Down>	cursor one line down, insert start column i_CTRL-G_<Down>	
CTRL-G j	cursor one line down, insert start column i_CTRL-G_j	
CTRL-G CTRL-J	cursor one line down, insert start column i_CTRL-G_CTRL-J	
<Left>	cursor one character left	i_<Left>
<Right>	cursor one character right	i_<Right>
<S-Left>	cursor one word back (like "b" command)	i_<S-Left>
<C-Left>	cursor one word back (like "b" command)	i_<C-Left>
<S-Right>	cursor one word forward (like "w" command) i_<S-Right>	
<C-Right>	cursor one word forward (like "w" command) i_<C-Right>	
<Home>	cursor to first char in the line	i_<Home>
<End>	cursor to after last char in the line	i_<End>
<C-Home>	cursor to first char in the file	i_<C-Home>
<C-End>	cursor to after last char in the file	i_<C-End>
<LeftMouse>	cursor to position of mouse click	i_<LeftMouse>
<S-Up>	move window one page up	i_<S-Up>
<PageUp>	move window one page up	i_<PageUp>
<S-Down>	move window one page down	i_<S-Down>
<PageDown>	move window one page down	i_<PageDown>
<ScrollWheelDown> move window three lines down	i_<ScrollWheelDown>	
<S-ScrollWheelDown> move window one page down	i_<S-ScrollWheelDown>	
<ScrollWheelUp> move window three lines up	i_<ScrollWheelUp>	
<S-ScrollWheelUp> move window one page up	i_<S-ScrollWheelUp>	
<ScrollWheelLeft> move window six columns left	i_<ScrollWheelLeft>	
<S-ScrollWheelLeft> move window one page left	i_<S-ScrollWheelLeft>	
<ScrollWheelRight> move window six columns right	i_<ScrollWheelRight>	
<S-ScrollWheelRight> move window one page right	i_<S-ScrollWheelRight>	
CTRL-O	execute one command, return to Insert mode i_CTRL-O	
CTRL-\ CTRL-O	like CTRL-O but don't move the cursor	i_CTRL-_CTRL-O
CTRL-G u	close undo sequence, start new change	i_CTRL-G_u
CTRL-G U	don't start a new undo block with the next i_CTRL-G_U left/right cursor movement, if the cursor stays within the same line	

The CTRL-O command sometimes has a side effect: If the cursor was beyond the end of the line, it will be put on the last character in the line. In mappings it's often better to use <Esc> (first put an "x" in the text, <Esc> will then always put the cursor on it). Or use CTRL-\ CTRL-O, but then beware of the cursor possibly being beyond the end of the line. Note that the command following CTRL-\ CTRL-O can still move the cursor, it is not restored to its original position.

The CTRL-O command takes you to Normal mode. If you then use a command enter Insert mode again it normally doesn't nest. Thus when typing "a<C-O>a" and then <Esc> takes you back to Normal mode, you do not need to type <Esc> twice. An exception is when not typing the command, e.g. when executing a mapping or sourcing a script. This makes mappings work that briefly switch to Insert mode.

The shifted cursor keys are not available on all terminals.

Another side effect is that a count specified before the "i" or "a" command is ignored. That is because repeating the effect of the command after CTRL-O is too complicated.

An example for using CTRL-G u:

```
:inoremap <C-H> <C-G>u<C-H>
```

This redefines the backspace key to start a new undo sequence. You can now
undo the effect of the backspace key, with CTRL-O u. Another example:

```
:inoremap <CR> <C-]><C-G>u<CR>
```

This starts a new undo block at each line break. It also expands
abbreviations before this.

An example for using CTRL-G U:

```
inoremap <Left>  <C-G>U<Left>
inoremap <Right> <C-G>U<Right>
inoremap <expr> <Home> col('.') == match(getline('.'), '\S') + 1 ?
 \ repeat('<C-G>U<Left>', col('.') - 1) :
 \ (col('.') < match(getline('.'), '\S') ?
 \     repeat('<C-G>U<Right>', match(getline('.'), '\S') + 0) :
 \     repeat('<C-G>U<Left>', col('.') - 1 - match(getline('.'), '\S')))
inoremap <expr> <End> repeat('<C-G>U<Right>', col('$') - col('.'))
inoremap ( ()<C-G>U<Left>
```

This makes it possible to use the cursor keys in Insert mode, without starting
a new undo block and therefore using . (redo) will work as expected. Also
entering a text like (with the "(" mapping from above):

   Lorem ipsum (dolor

will be repeatable by using . to the expected

   Lorem ipsum (dolor)

Cursor keys are captured by . (ins-repeat), so these mappings are only
needed to keep the cursor-move from starting a new undo block.

Using CTRL-O splits undo: the text typed before and after it is undone
separately. If you want to avoid this (e.g., in a mapping) you might be able
to use CTRL-R = i_CTRL-R. E.g., to call a function:

```
:imap <F2> <C-R>=MyFunc()<CR>
```

When the 'whichwrap' option is set appropriately, the <Left> and <Right>
keys on the first/last character in the line make the cursor wrap to the
previous/next line.

The CTRL-G j and CTRL-G k commands can be used to insert text in front of a
column. Example:

```
int i;
int j;
```

Position the cursor on the first "int", type "istatic <C-G>j   ". The
result is:

```
static int i;
       int j;
```

When inserting the same text in front of the column in every line, use the
Visual blockwise command "I" v_b_I.

3. 'textwidth' and 'wrapmargin' options ins-textwidth

The 'textwidth' option can be used to automatically break a line before it
gets too long. Set the 'textwidth' option to the desired maximum line
length. If you then type more characters (not spaces or tabs), the
last word will be put on a new line (unless it is the only word on the
line). If you set 'textwidth' to 0, this feature is disabled.

The 'wrapmargin' option does almost the same. The difference is that
'textwidth' has a fixed width while 'wrapmargin' depends on the width of the
screen. When using 'wrapmargin' this is equal to using 'textwidth' with a
value equal to (columns - 'wrapmargin'), where columns is the width of the
screen.

When 'textwidth' and 'wrapmargin' are both set, 'textwidth' is used.

If you don't really want to break the line, but view the line wrapped at a
convenient place, see the 'linebreak' option.

The line is only broken automatically when using Insert mode, or when
appending to a line. When in replace mode and the line length is not
changed, the line will not be broken.

Long lines are broken if you enter a non-white character after the margin.
The situations where a line will be broken can be restricted by adding
characters to the 'formatoptions' option:
"l" Only break a line if it was not longer than 'textwidth' when the insert
     started.
"v" Only break at a white character that has been entered during the

current insert command. This is mostly Vi-compatible.
"lv" Only break if the line was not longer than 'textwidth' when the insert started and only at a white character that has been entered during the current insert command. Only differs from "l" when entering non-white characters while crossing the 'textwidth' boundary.

Normally an internal function will be used to decide where to break the line. If you want to do it in a different way set the 'formatexpr' option to an expression that will take care of the line break.

If you want to format a block of text, you can use the "gq" operator. Type "gq" and a movement command to move the cursor to the end of the block. In many cases, the command "gq}" will do what you want (format until the end of paragraph). Alternatively, you can use "gqap", which will format the whole paragraph, no matter where the cursor currently is. Or you can use Visual mode: hit "v", move to the end of the block, and type "gq". See also gg.

4. 'expandtab', 'softtabstop' and 'smarttab' options ins-expandtab

If the 'expandtab' option is on, spaces will be used to fill the amount of whitespace of the tab. If you want to enter a real <Tab>, type CTRL-V first (use CTRL-Q when CTRL-V is mapped i_CTRL-Q).
The 'expandtab' option is off by default. Note that in Replace mode, a single character is replaced with several spaces. The result of this is that the number of characters in the line increases. Backspacing will delete one space at a time. The original character will be put back for only one space that you backspace over (the last one).

ins-softtabstop

When the 'softtabstop' option is non-zero, a <Tab> inserts 'softtabstop' positions, and a <BS> used to delete white space, will delete 'softtabstop' positions. This feels like 'tabstop' was set to 'softtabstop', but a real <Tab> character still takes 'tabstop' positions, so your file will still look correct when used by other applications.

If 'softtabstop' is non-zero, a <BS> will try to delete as much white space to move to the previous 'softtabstop' position, except when the previously inserted character is a space, then it will only delete the character before the cursor. Otherwise you cannot always delete a single character before the cursor. You will have to delete 'softtabstop' characters first, and then type extra spaces to get where you want to be.

ins-smarttab

When the 'smarttab' option is on, the <Tab> key indents by 'shiftwidth' if the cursor is in leading whitespace. The <BS> key has the opposite effect. This behaves as if 'softtabstop' were set to the value of 'shiftwidth'. This option allows the user to set 'softtabstop' to a value other than 'shiftwidth' and still use the <Tab> key for indentation.

5. Replace mode Replace Replace-mode mode-replace

Enter Replace mode with the "R" command in normal mode.

In Replace mode, one character in the line is deleted for every character you type. If there is no character to delete (at the end of the line), the typed character is appended (as in Insert mode). Thus the number of characters in a line stays the same until you get to the end of the line. If a <NL> is typed, a line break is inserted and no character is deleted.

Be careful with <Tab> characters. If you type a normal printing character in its place, the number of characters is still the same, but the number of columns will become smaller.

If you delete characters in Replace mode (with <BS>, CTRL-W, or CTRL-U), what happens is that you delete the changes. The characters that were replaced are restored. If you had typed past the existing text, the characters you added are deleted. This is effectively a character-at-a-time undo.

If the 'expandtab' option is on, a <Tab> will replace one character with several spaces. The result of this is that the number of characters in the line increases. Backspacing will delete one space at a time. The original character will be put back for only one space that you backspace over (the last one).

6. Virtual Replace mode vreplace-mode Virtual-Replace-mode

Enter Virtual Replace mode with the "gR" command in normal mode.

Virtual Replace mode is similar to Replace mode, but instead of replacing actual characters in the file, you are replacing screen real estate, so that characters further on in the file never appear to move.

So if you type a <Tab> it may replace several normal characters, and if you type a letter on top of a <Tab> it may not replace anything at all, since the <Tab> will still line up to the same place as before.

Typing a <NL> still doesn't cause characters later in the file to appear to move. The rest of the current line will be replaced by the <NL> (that is, they are deleted), and replacing continues on the next line. A new line is NOT inserted unless you go past the end of the file.

Interesting effects are seen when using CTRL-T and CTRL-D. The characters before the cursor are shifted sideways as normal, but characters later in the

line still remain still. CTRL-T will hide some of the old line under the shifted characters, but CTRL-D will reveal them again.

As with Replace mode, using <BS> etc will bring back the characters that were replaced. This still works in conjunction with 'smartindent', CTRL-T and CTRL-D, 'expandtab', 'smarttab', 'softtabstop', etc.

In 'list' mode, Virtual Replace mode acts as if it was not in 'list' mode, unless "L" is in 'cpoptions'.

Note that the only situations for which characters beyond the cursor should appear to move are in List mode 'list', and occasionally when 'wrap' is set (and the line changes length to become shorter or wider than the width of the screen). In other cases spaces may be inserted to avoid following characters to move.

This mode is very useful for editing <Tab> separated columns in tables, for entering new data while keeping all the columns aligned.

7. Insert mode completion

ins-completion

In Insert and Replace mode, there are several commands to complete part of a keyword or line that has been typed. This is useful if you are using complicated keywords (e.g., function names with capitals and underscores).

Completion can be done for:

1.	Whole lines	i_CTRL-X_CTRL-L
2.	keywords in the current file	i_CTRL-X_CTRL-N
3.	keywords in 'dictionary'	i_CTRL-X_CTRL-K
4.	keywords in 'thesaurus', thesaurus-style	i_CTRL-X_CTRL-T
5.	keywords in the current and included files	i_CTRL-X_CTRL-I
6.	tags	i_CTRL-X_CTRL-]
7.	file names	i_CTRL-X_CTRL-F
8.	definitions or macros	i_CTRL-X_CTRL-D
9.	Vim command-line	i_CTRL-X_CTRL-V
10.	User defined completion	i_CTRL-X_CTRL-U
11.	omni completion	i_CTRL-X_CTRL-O
12.	Spelling suggestions	i_CTRL-X_s
13.	completions from 'complete'	i_CTRL-N i_CTRL-P
14.	contents from registers	i_CTRL-X_CTRL-R

Additionally, i_CTRL-X_CTRL-Z stops completion without changing the text.

All these, except CTRL-N and CTRL-P, are done in CTRL-X mode. This is a sub-mode of Insert and Replace modes. You enter CTRL-X mode by typing CTRL-X and one of the CTRL-X commands. You exit CTRL-X mode by typing a key that is not a valid CTRL-X mode command. Valid keys are the CTRL-X command itself, CTRL-N (next), and CTRL-P (previous).

By default, the possible completions are showed in a menu and the first completion is inserted into the text. This can be adjusted with 'completeopt'.

To get the current completion information, complete_info() can be used. Also see the 'infercase' option if you want to adjust the case of the match.

When inserting a selected candidate word from the popup-menu, the part of the candidate word that does not match the query is highlighted using hl-ComplMatchIns. If fuzzy is enabled in 'completeopt', highlighting will not be applied.

complete_CTRL-E

When completion is active you can use CTRL-E to stop it and go back to the originally typed text. The CTRL-E will not be inserted.

complete_CTRL-Y

When the popup menu is displayed, CTRL-Y stops completion and accepts the currently selected entry. Typing a space, Enter, or some other unprintable character will leave completion mode and insert that typed character. If you want to use <Enter> to accept a completion item, use this mapping:

```
inoremap <expr> <cr> pumvisible() ? '<c-y>' : '<cr>'
```

When the popup menu is displayed there are a few more special keys, see popupmenu-keys.

Note: The keys that are valid in CTRL-X mode are not mapped. This allows for :map <C-F> <C-X><C-F> to work. The key that ends CTRL-X mode (any key that is not a valid CTRL-X mode command) is mapped. Also, when doing completion with 'complete' or when completion was started with complete() mappings apply as usual.

E565

Note: While completion is active Insert mode can't be used recursively and buffer text cannot be changed. Mappings that somehow invoke ":normal i.." will generate an E565 error.

The following mappings are suggested to make typing the completion commands a bit easier (although they will hide other commands):

```
:inoremap <C-]> <C-X><C-]>
:inoremap <C-F> <C-X><C-F>
:inoremap <C-D> <C-X><C-D>
:inoremap <C-L> <C-X><C-L>
```

As a special case, typing CTRL-R to perform register insertion (see i_CTRL-R) will not exit CTRL-X mode. This is primarily to allow the use of the '=' register to call some function to determine the next operation. If the contents of the register (or result of the '=' register evaluation) are not valid CTRL-X mode keys, then CTRL-X mode will be exited as if those keys had been typed.

For example, the following will map <Tab> to either actually insert a <Tab> if the current line is currently only whitespace, or start/continue a CTRL-N completion operation:

```
function _G.clever_tab()
  if vim.fn.getline('.'):sub(1, vim.fn.col('.') - 1):match('^%s*$') then
    return vim.keycode('<Tab>')
  else
    return vim.keycode('<C-N>')
  end
end
vim.keymap.set('i', '<Tab>', '<C-R>=v:lua.clever_tab()<CR>')
```

Completing whole lines compl-whole-line

i_CTRL-X_CTRL-L

CTRL-X CTRL-L Search backwards for a line that starts with the same characters as those in the current line before the cursor. Indent is ignored. The matching line is inserted in front of the cursor.
The 'complete' option is used to decide which buffers are searched for a match. Both loaded and unloaded buffers are used.

CTRL-L or CTRL-P Search backwards for next matching line. This line replaces the previous matching line.

CTRL-N Search forward for next matching line. This line replaces the previous matching line.

CTRL-X CTRL-L After expanding a line you can additionally get the line next to it by typing CTRL-X CTRL-L again, unless a double CTRL-X is used. Only works for loaded buffers.

Completing keywords in current file compl-current

i_CTRL-X_CTRL-P

i_CTRL-X_CTRL-N

CTRL-X CTRL-N Search forwards for words that start with the keyword in front of the cursor. The found keyword is inserted in front of the cursor.

CTRL-X CTRL-P Search backwards for words that start with the keyword in front of the cursor. The found keyword is inserted in front of the cursor.

CTRL-N Search forward for next matching keyword. This keyword replaces the previous matching keyword.

CTRL-P Search backwards for next matching keyword. This keyword replaces the previous matching keyword.

	CTRL-X CTRL-N or
	CTRL-X CTRL-P	Further use of CTRL-X CTRL-N or CTRL-X CTRL-P will
			copy the words following the previous expansion in
			other contexts unless a double CTRL-X is used.

If there is a keyword in front of the cursor (a name made out of alphabetic characters and characters in 'iskeyword'), it is used as the search pattern, with "\<" prepended (meaning: start of a word). Otherwise "\<\k\k" is used as search pattern (start of any keyword of at least two characters).

In Replace mode, the number of characters that are replaced depends on the length of the matched string. This works like typing the characters of the matched string in Replace mode.

If there is not a valid keyword character before the cursor, any keyword of at least two characters is matched.

	e.g., to get:
	    printf("(%g, %g, %g)", vector[0], vector[1], vector[2]);
	just type:
	    printf("(%g, %g, %g)", vector[0], ^P[1], ^P[2]);

The search wraps around the end of the file, the value of 'wrapscan' is not used here.

Multiple repeats of the same completion are skipped; thus a different match will be inserted at each CTRL-N and CTRL-P (unless there is only one matching keyword).

Single character matches are never included, as they usually just get in the way of what you were really after.

	e.g., to get:
		printf("name = %s\n", name);
	just type:
		printf("name = %s\n", n^P);
	or even:
		printf("name = %s\n", ^P);

The 'n' in '\n' is skipped.

After expanding a word, you can use CTRL-X CTRL-P or CTRL-X CTRL-N to get the word following the expansion in other contexts. These sequences search for the text just expanded and further expand by getting an extra word. This is useful if you need to repeat a sequence of complicated words. Although CTRL-P and CTRL-N look just for strings of at least two characters, CTRL-X CTRL-P and CTRL-X CTRL-N can be used to expand words of just one character.

	e.g., to get:
		México
	you can type:
		M^N^P^X^P^X^P

CTRL-N starts the expansion and then CTRL-P takes back the single character "M", the next two CTRL-X CTRL-P's get the words "&eacute" and ";xico".

If the previous expansion was split, because it got longer than 'textwidth', then just the text in the current line will be used.

If the match found is at the end of a line, then the first word in the next line will be inserted and the message "Word from other line" displayed, if this word is accepted the next CTRL-X CTRL-P or CTRL-X CTRL-N will search for those lines starting with this word.

Completing keywords in 'dictionary'	compl-dictionary
	i_CTRL-X_CTRL-K

CTRL-X CTRL-K		Search the files given with the 'dictionary' option
			for words that start with the keyword in front of the
			cursor. This is like CTRL-N, but only the dictionary
			files are searched, not the current file. The found
			keyword is inserted in front of the cursor. This
			could potentially be pretty slow, since all matches
			are found before the first match is used. By default,
			the 'dictionary' option is empty.
			For suggestions where to find a list of words, see the
			'dictionary' option.
			'ignorecase', 'smartcase' and 'infercase' apply.

	CTRL-K or
	CTRL-N		Search forward for next matching keyword. This
			keyword replaces the previous matching keyword.

	CTRL-P		Search backwards for next matching keyword. This
			keyword replaces the previous matching keyword.

Completing words in 'thesaurus'	compl-thesaurus
	i_CTRL-X_CTRL-T

CTRL-X CTRL-T		Works as CTRL-X CTRL-K, but in a special way. It uses
			the 'thesaurus' option instead of 'dictionary'. If a
			match is found in the thesaurus file, all the
			remaining words on the same line are included as
			matches, even though they don't complete the word.
			Thus a word can be completely replaced.

	CTRL-T or
	CTRL-N		Search forward for next matching keyword. This
			keyword replaces the previous matching keyword.

	CTRL-P		Search backwards for next matching keyword. This
			keyword replaces the previous matching keyword.

In the file used by the 'thesaurus' option each line in the file should
contain words with similar meaning, separated by non-keyword characters (white
space is preferred). Maximum line length is 510 bytes.

For an example, imagine the 'thesaurus' file has a line like this:

```
angry furious mad enraged
```

Placing the cursor after the letters "ang" and typing CTRL-X CTRL-T would
complete the word "angry"; subsequent presses would change the word to
"furious", "mad" etc.

Other uses include translation between two languages, or grouping API
functions by keyword.

An English word list was added to this github issue:
https://github.com/vim/vim/issues/629#issuecomment-443293282
Unpack thesaurus_pkg.zip, put the thesaurus.txt file somewhere, e.g.
~/.vim/thesaurus/english.txt, and the 'thesaurus' option to this file name.

Completing keywords with 'thesaurusfunc' compl-thesaurusfunc

If the 'thesaurusfunc' option is set, then the user specified function is
invoked to get the list of completion matches and the 'thesaurus' option is
not used. See complete-functions for an explanation of how the function is
invoked and what it should return.

Here is an example that uses the "aiksaurus" command (provided by Magnus
Groß):

```
vim.o.thesaurusfunc = function(findstart, base)
  if findstart == 1 then
    return vim.fn.searchpos('\\<', 'bnW', vim.fn.line('.'))[2] - 1
  end
  local res = {}
  local h = ''
  for _, l in ipairs(vim.fn.systemlist('aiksaurus '
                                       .. vim.fn.shellescape(base))) do
    if l:sub(1, 4) == '=== ' then
      h = ('(%s)'):format((l:sub(5):gsub(' =*$', '')))
    elseif l == 'Alphabetically similar known words are: ' then
      h = '🔮'
    elseif l:match('^%a') or (h == '🔮' and l:sub(1, 1) == '\t') then
      local words = (l:gsub('^\t', ''))
      for _, val in ipairs(vim.split(words, ', ', { plain = true })) do
        table.insert(res, { word = val, menu = h })
      end
    end
  end
  return res
end
```

Completing keywords in the current and included files compl-keyword

The 'include' option is used to specify a line that contains an include file
name. The 'path' option is used to search for include files.

i_CTRL-X_CTRL-I

CTRL-X CTRL-I Search for the first keyword in the current and
               included files that starts with the same characters
               as those before the cursor. The matched keyword is
               inserted in front of the cursor.

       CTRL-N Search forwards for next matching keyword. This
               keyword replaces the previous matching keyword.
               Note: CTRL-I is the same as <Tab>, which is likely to
               be typed after a successful completion, therefore
               CTRL-I is not used for searching for the next match.

       CTRL-P Search backward for previous matching keyword. This
               keyword replaces the previous matching keyword.

       CTRL-X CTRL-I Further use of CTRL-X CTRL-I will copy the words
               following the previous expansion in other contexts
               unless a double CTRL-X is used.

Completing tags compl-tag

i_CTRL-X_CTRL-]

CTRL-X CTRL-] Search for the first tag that starts with the same
               characters as before the cursor. The matching tag is
               inserted in front of the cursor. Alphabetic
               characters and characters in 'iskeyword' are used
               to decide which characters are included in the tag
               name (same as for a keyword). See also CTRL-].
               The 'showfulltag' option can be used to add context
               from around the tag definition.
       CTRL-] or
       CTRL-N Search forwards for next matching tag. This tag
               replaces the previous matching tag.

CTRL-P Search backward for previous matching tag. This tag replaces the previous matching tag.

Completing file names compl-filename

i_CTRL-X_CTRL-F

CTRL-X CTRL-F Search for the first file name that starts with the same characters as before the cursor. The matching file name is inserted in front of the cursor. Alphabetic characters and characters in 'isfname' are used to decide which characters are included in the file name. **Note:** the 'path' option is not used here (yet).

CTRL-F or
CTRL-N Search forwards for next matching file name. This file name replaces the previous matching file name.

CTRL-P Search backward for previous matching file name. This file name replaces the previous matching file name.

Completing definitions or macros compl-define

The 'define' option is used to specify a line that contains a definition. The 'include' option is used to specify a line that contains an include file name. The 'path' option is used to search for include files.

i_CTRL-X_CTRL-D

CTRL-X CTRL-D Search in the current and included files for the first definition (or macro) name that starts with the same characters as before the cursor. The found definition name is inserted in front of the cursor.

CTRL-D or
CTRL-N Search forwards for next matching macro name. This macro name replaces the previous matching macro name.

CTRL-P Search backward for previous matching macro name. This macro name replaces the previous matching macro name.

CTRL-X CTRL-D Further use of CTRL-X CTRL-D will copy the words following the previous expansion in other contexts unless a double CTRL-X is used.

Completing Vim commands compl-vim

Completion is context-sensitive. It works like on the Command-line. It completes an Ex command as well as its arguments. This is useful when writing a Vim script.

i_CTRL-X_CTRL-V

CTRL-X CTRL-V Guess what kind of item is in front of the cursor and find the first match for it.
Note: When CTRL-V is mapped you can often use CTRL-Q instead of i_CTRL-Q.

CTRL-V or
CTRL-N Search forwards for next match. This match replaces the previous one.

CTRL-P Search backwards for previous match. This match replaces the previous one.

CTRL-X CTRL-V Further use of CTRL-X CTRL-V will do the same as CTRL-V. This allows mapping a key to do Vim command completion, for example:

```
:imap <Tab> <C-X><C-V>
```

Completing contents from registers compl-register-words

i_CTRL-X_CTRL-R

CTRL-X CTRL-R Guess what kind of item is in front of the cursor from all registers and find the first match for it.
Further use of CTRL-R (without CTRL-X) will insert the register content, see i_CTRL-R.
'ignorecase' applies to the matching.

CTRL-N Search forwards for next match. This match replaces the previous one.

CTRL-P Search backwards for previous match. This match replaces the previous one.

CTRL-X CTRL-R Further use of CTRL-X CTRL-R will copy the line following the previous expansion in other contexts unless a double CTRL-X is used (e.g. this switches from completing register words to register contents).

User defined completion compl-function

Completion is done by a function that can be defined by the user with the 'completefunc' option. See below for how the function is called and an example complete-functions.

i_CTRL-X_CTRL-U

CTRL-X CTRL-U Guess what kind of item is in front of the cursor and find the first match for it.

CTRL-U or
CTRL-N Use the next match. This match replaces the previous one.

CTRL-P Use the previous match. This match replaces the previous one.

Omni completion omnicompletion compl-omni

Completion is done by a function that can be defined by the user with the 'omnifunc' option. This is to be used for filetype-specific completion.

See below for how the function is called and an example complete-functions. For remarks about specific filetypes see compl-omni-filetypes.
More completion scripts will appear, check www.vim.org. Currently there is a first version for C++.

i_CTRL-X_CTRL-O

CTRL-X CTRL-O Guess what kind of item is in front of the cursor and find the first match for it.

CTRL-O or
CTRL-N Use the next match. This match replaces the previous one.

CTRL-P Use the previous match. This match replaces the previous one.

Spelling suggestions compl-spelling

A word before or at the cursor is located and correctly spelled words are suggested to replace it. If there is a badly spelled word in the line, before or under the cursor, the cursor is moved to after it. Otherwise the word just before the cursor is used for suggestions, even though it isn't badly spelled.

NOTE: CTRL-S suspends display in many Unix terminals. Use 's' instead. Type CTRL-Q to resume displaying.

i_CTRL-X_CTRL-S i_CTRL-X_s

CTRL-X CTRL-S or
CTRL-X s Locate the word in front of the cursor and find the first spell suggestion for it.

CTRL-S or
CTRL-N Use the next suggestion. This replaces the previous one. Note that you can't use 's' here.

CTRL-P Use the previous suggestion. This replaces the previous one.

Completing from different sources compl-generic

i_CTRL-N

CTRL-N Find the next match for a word ending at the cursor, using the sources specified in the 'complete' option. All sources complete from keywords, except functions, which may complete from non-keyword. The matched text is inserted before the cursor.

i_CTRL-P

CTRL-P Same as CTRL-N, but find the previous match.

CTRL-N Search forward through the matches and insert the next one.

CTRL-P Search backward through the matches and insert the previous one.

CTRL-X CTRL-N or
CTRL-X CTRL-P Further use of CTRL-X CTRL-N or CTRL-X CTRL-P will copy the words following the previous expansion in other contexts unless a double CTRL-X is used.

Stop completion compl-stop

i_CTRL-X_CTRL-Z

CTRL-X CTRL-Z Stop completion without changing the text.

AUTOCOMPLETION ins-autocompletion

Vim can display a completion menu as you type, similar to using i_CTRL-N, but triggered automatically. See 'autocomplete'. The menu items are collected from the sources listed in the 'complete' option, in order.

A decaying timeout keeps Vim responsive. Sources earlier in the 'complete' list get more time (higher priority), but all sources receive at least a small time slice.

This mode is fully compatible with other completion modes. You can invoke any of them at any time by typing CTRL-X, which temporarily suspends autocompletion. To use i_CTRL-N or i_CTRL-X_CTRL-N specifically, press CTRL-E first to dismiss the popup menu (see complete_CTRL-E).

ins-autocompletion-example

Example setup

A typical configuration for automatic completion with a popup menu:

```
set autocomplete
set complete=.^5,w^5,b^5,u^5
set completeopt=popup

inoremap <silent><expr> <Tab>   pumvisible() ? "\<C-n>" : "\<Tab>"
inoremap <silent><expr> <S-Tab> pumvisible() ? "\<C-p>" : "\<S-Tab>"
```

This enables automatic completion with suggestions from the current buffer, other windows, and listed buffers, displayed in a popup menu. Each source is limited to 5 candidates. <Tab> and <S-Tab> move through the items when the menu is visible. Optionally, add "preinsert" to 'completeopt' to insert the longest common prefix automatically. You can also add other completion sources to 'complete' as needed.

See also 'autocomplete', 'autocompletedelay' and 'autocompletetimeout'.

For command-line autocompletion, see cmdline-autocompletion.

To get LSP-driven auto-completion, see lsp-completion.

FUNCTIONS FOR FINDING COMPLETIONS **complete-functions**

This applies to 'completefunc', 'thesaurusfunc' and 'omnifunc'.

The function is called in two different ways:

- First the function is called to find the start of the text to be completed.

- Later the function is called to actually find the matches.

On the first invocation the arguments are:
```
a:findstart  1
a:base       empty
```

The function must return the column where the completion starts. It must be a number between zero and the cursor column "col('.')". This involves looking at the characters just before the cursor and including those characters that could be part of the completed item. The text between this column and the cursor column will be replaced with the matches. If the returned value is larger than the cursor column, the cursor column is used.

Negative return values:
```
-2   To cancel silently and stay in completion mode.
-3   To cancel silently and leave completion mode.
Another negative value: completion starts at the cursor column
```

On the second invocation the arguments are:
```
a:findstart  0
a:base       the text with which matches should match; the text that was
             located in the first call (can be empty)
```

The function must return a List with the matching words. These matches usually include the "a:base" text. When there are no matches return an empty List. Note that the cursor may have moved since the first invocation, the text may have been changed.

In order to return more information than the matching words, return a Dict that contains the List. The Dict can have these items:
```
        words           The List of matching words (mandatory).
        refresh         A string to control re-invocation of the function
                        (optional).
                        The only value currently recognized is "always", the
                        effect is that the function is called whenever the
                        leading text is changed.
```
Other items are ignored.

For acting upon end of completion, see the CompleteDonePre and CompleteDone autocommand event.

For example, the function can contain this:

```
let matches = ... list of words ...
return {'words': matches, 'refresh': 'always'}
```

If looking for matches is time-consuming, complete_check() may be used to maintain responsiveness.

complete-items

Each list item can either be a string or a Dictionary. When it is a string it is used as the completion. When it is a Dictionary it can contain these items:
```
        word            the text that will be inserted, mandatory
        abbr            abbreviation of "word"; when not empty it is used in
                        the menu instead of "word"
        menu            extra text for the popup menu, displayed after "word"
                        or "abbr"
        info            more information about the item, can be displayed in a
                        preview window
        kind            single letter indicating the type of completion
        icase           when non-zero case is to be ignored when comparing
                        items to be equal; when omitted zero is used, thus
                        items that only differ in case are added
```

```
        equal           when non-zero, always treat this item to be equal when
                        comparing.  Which means, "equal=1" disables filtering
                        of this item.
        dup             when non-zero this match will be added even when an
                        item with the same word is already present.
        empty           when non-zero this match will be added even when it is
                        an empty string
        user_data       custom data which is associated with the item and
                        available in v:completed_item; it can be any type;
                        defaults to an empty string
        abbr_hlgroup    an additional highlight group whose attributes are
                        combined with hl-PmenuSel and hl-Pmenu or
                        hl-PmenuMatchSel and hl-PmenuMatch highlight
                        attributes in the popup menu to apply cterm and gui
                        properties (with higher priority) like strikethrough
                        to the completion items abbreviation.
                        The group can be defined later.
        kind_hlgroup    an additional highlight group specifically for setting
                        the highlight attributes of the completion kind.  When
                        this field is present, it will override the
                        hl-PmenuKind highlight group, allowing for the
                        customization of ctermfg and guifg properties for the
                        completion kind.
                        The group can be defined later.
        match           See "matches" in complete_info().
        preselect       when non-zero this item is selected by default in the
                        popup menu. Only if 'completeopt' has "preselect". If
                        multiple items have "preselect" set, the first one is
                        used.
        commit_chars    Characters that accept the completion when typed while
                        the item is selected: the match is accepted as with
                        complete_CTRL-Y, then the character is inserted as
                        usual.  Takes precedence over extending the leader.
                        Only while the popup menu is displayed.  Control
                        characters (below 0x20) are ignored.

All of these except "icase", "equal", "dup" and "empty" must be a string.  If
an item does not meet these requirements then an error message is given and
further items in the list are not used.  You can mix string and Dictionary
items in the returned list.

The "menu" item is used in the popup menu and may be truncated, thus it should
be relatively short.  The "info" item can be longer, it will  be displayed in
the preview window when "preview" appears in 'completeopt'.  The "info" item
will also remain displayed after the popup menu has been removed.  This is
useful for function arguments.  Use a single space for "info" to remove
existing text in the preview window.  The size of the preview window is three
lines, but 'previewheight' is used when it has a value of 1 or 2.

The "kind" item uses a single letter to indicate the kind of completion.  This
may be used to show the completion differently (different color or icon).
Currently these types can be used:
        v       variable
        f       function or method
        m       member of a struct or class
        t       typedef
        d       #define or macro

When searching for matches takes some time call complete_add() to add each
match to the total list.  These matches should then not appear in the returned
list!  Call complete_check() now and then to allow the user to press a key
while still searching for matches.  Stop searching when it returns non-zero.

                                                                        E840

The function is allowed to move the cursor, it is restored afterwards.
The function is not allowed to move to another window or delete text.

An example that completes the names of the months:
```

```lua
vim.o.completefunc = function(findstart, base)
  if findstart == 1 then -- Locate the start of the word.
    local line = vim.fn.getline('.')
    local start = vim.fn.col('.') - 1
    while start > 0 and line:sub(start, start):match('%a') do
      start = start - 1
    end
    return start
  else -- Find months matching with "base".
    local res = {}
    local months = 'Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec'
    for m in months:gmatch('%a+') do
      if vim.startswith(m, base) then
        table.insert(res, m)
      end
    end
    return res
  end
end
```

```
The same, but now pretending searching for matches is slow:
```

```
vim.o.completefunc = function(findstart, base)
  if findstart == 1 then -- Locate the start of the word.
    local line = vim.fn.getline('.')
    local start = vim.fn.col('.') - 1
    while start > 0 and line:sub(start, start):match('%a') do
      start = start - 1
    end
    return start
  else -- Find months matching with "base".
    local months = 'Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec'
    for m in months:gmatch('%a+') do
      if vim.startswith(m, base) then
        vim.fn.complete_add(m)
      end
      vim.uv.sleep(300) -- Simulate searching for next match.
      if vim.fn.complete_check() ~= 0 then
        break
      end
    end
    return {}
  end
end
```

INSERT COMPLETION POPUP MENU **ins-completion-menu**

popupmenu-completion

Vim can display the matches in a popup menu.

The menu is used when:
- The 'completeopt' option contains "menu" or "menuone".

- There are at least two matches. One if "menuone" is used.

The 'pumheight' option can be used to set a maximum height. The default is to use all space available.
The 'pumwidth' option can be used to set a minimum width. The default is 15 characters.

compl-states

There are three states:
1. A complete match has been inserted, e.g., after using CTRL-N or CTRL-P.

2. A cursor key has been used to select another match. The match was not inserted then, only the entry in the popup menu is highlighted.
3. Only part of a match has been inserted and characters were typed or the backspace key was used. The list of matches was then adjusted for what is in front of the cursor.

You normally start in the first state, with the first match being inserted. When "longest" is in 'completeopt' and there is more than one match you start in the third state.

If you select another match, e.g., with CTRL-N or CTRL-P, you go to the first state. This doesn't change the list of matches.

When you are back at the original text then you are in the third state. To get there right away you can use a mapping that uses CTRL-P right after starting the completion:

```
:imap <F7> <C-N><C-P>
```

popupmenu-keys

In the first state these keys have a special meaning:
<BS> and CTRL-H Delete one character, find the matches for the word before the cursor. This reduces the list of matches, often to one entry, and switches to the second state. When 'autocomplete' is active, clears the selected item in the menu.
Any non-special character:
  Stop completion without changing the match and insert the typed character.

In the second and third state these keys have a special meaning:
<BS> and CTRL-H Delete one character, find the matches for the shorter word before the cursor. This may find more matches. When 'autocomplete' is active, clears the selected item in the menu.
CTRL-L Add one character from the current match, may reduce the number of matches.
any printable, non-white character:
  Add this character and reduce the number of matches.

In all three states these can be used:

CTRL-Y	Yes: Accept the currently selected match and stop completion.
CTRL-E	End completion, go back to what was there before selecting a match (what was typed or longest common string).
<PageUp>	Select a match several entries back, but don't insert it.
<PageDown>	Select a match several entries further, but don't insert it.
<Up>	Select the previous match, as if CTRL-P was used, but don't insert it.
<Down>	Select the next match, as if CTRL-N was used, but don't insert it.
<Space> or <Tab>	Stop completion without changing the match and insert the typed character.

The behavior of the <Enter> key depends on the state you are in:

first state:	Use the text as it is and insert a line break.
second state:	Insert the currently selected match.
third state:	Use the text as it is and insert a line break.

In other words: If you used the cursor keys to select another entry in the list of matches then the <Enter> key inserts that match. If you typed something else then <Enter> inserts a line break.

The colors of the menu can be changed with these highlight groups:

Pmenu	normal item	hl-Pmenu
PmenuSel	selected item	hl-PmenuSel
PmenuSbar	scrollbar	hl-PmenuSbar
PmenuThumb	thumb of the scrollbar	hl-PmenuThumb

There are no special mappings for when the popup menu is visible. However, you can use an Insert mode mapping that checks the pumvisible() function to do something different. Example:

```
:inoremap <Down> <C-R>=pumvisible() ? "\<lt>C-N>" : "\<lt>Down>"<CR>
```

You can use of <expr> in mapping to have the popup menu used when typing a character and some condition is met. For example, for typing a dot:

```
inoremap <expr> . MayComplete()
func MayComplete()
    if (can complete)
      return ".\<C-X>\<C-O>"
    endif
    return '.'
endfunc
```

See :map-<expr> for more info.

FILETYPE-SPECIFIC REMARKS FOR OMNI COMPLETION compl-omni-filetypes

The file used for {filetype} should be autoload/{filetype}complete.vim in 'runtimepath'. Thus for "java" it is autoload/javacomplete.vim.

C ft-c-omni

Completion of C code requires a tags file. You should use Universal/Exuberant ctags, because it adds extra information that is needed for completion. You can find it here:

Universal Ctags: https://ctags.io

Universal Ctags is preferred, Exuberant Ctags is no longer maintained.

If you want to complete system functions you can do something like this. Use ctags to generate a tags file for all the system header files:

```
% ctags -R -f ~/.config/nvim/systags /usr/include /usr/local/include
```

In your vimrc file add this tags file to the 'tags' option:

```
set tags+=~/.config/nvim/systags
```

When using CTRL-X CTRL-O after a name without any "." or "->" it is completed from the tags file directly. This works for any identifier, also function names. If you want to complete a local variable name, which does not appear in the tags file, use CTRL-P instead.

When using CTRL-X CTRL-O after something that has "." or "->" Vim will attempt to recognize the type of the variable and figure out what members it has. This means only members valid for the variable will be listed.

When a member name already was complete, CTRL-X CTRL-O will add a "." or "->" for composite types.

Vim doesn't include a C compiler, only the most obviously formatted declarations are recognized. Preprocessor stuff may cause confusion. When the same structure name appears in multiple places all possible members are included.

CSS ft-css-omni

Complete properties and their appropriate values according to CSS 2.1 specification.

HTML ft-html-omni

XHTML ft-xhtml-omni

CTRL-X CTRL-O provides completion of various elements of (X)HTML files. It is designed to support writing of XHTML 1.0 Strict files but will also work for other versions of HTML. Features:

- after "<" complete tag name depending on context (no div suggestion inside of an a tag); '/>' indicates empty tags
- inside of tag complete proper attributes (no width attribute for an a tag); show also type of attribute; "*" indicates required attributes
- when attribute has limited number of possible values help to complete them
- complete names of entities
- complete values of "class" and "id" attributes with data obtained from <style> tag and included CSS files
- when completing value of "style" attribute or working inside of "style" tag switch to ft-css-omni completion
- when completing values of events attributes or working inside of "script" tag switch to ft-javascript-omni completion
- when used after "</" CTRL-X CTRL-O will close the last opened tag

Note: When used first time completion menu will be shown with little delay
- this is time needed for loading of data file.

Note: Completion may fail in badly formatted documents. In such case try to run :make command to detect formatting problems.

HTML flavor html-flavor

The default HTML completion depends on the filetype. For HTML files it is HTML 4.01 Transitional ('filetype' is "html"), for XHTML it is XHTML 1.0 Strict ('filetype' is "xhtml").

When doing completion outside of any other tag you will have possibility to choose DOCTYPE and the appropriate data file will be loaded and used for all next completions.

More about format of data file in xml-omni-datafile. Some of the data files may be found on the Vim website (www).

Note that b:html_omni_flavor may point to a file with any XML data. This makes possible to mix PHP (ft-php-omni) completion with any XML dialect (assuming you have data file for it). Without setting that variable XHTML 1.0 Strict will be used.

JAVASCRIPT ft-javascript-omni

Completion of most elements of JavaScript language and DOM elements.

Complete:

- variables
- function name; show function arguments
- function arguments
- properties of variables trying to detect type of variable
- complete DOM objects and properties depending on context
- keywords of language

Completion works in separate JavaScript files (&ft==javascript), inside of <script> tag of (X)HTML and in values of event attributes (including scanning of external files).

DOM compatibility

At the moment (beginning of 2006) there are two main browsers - MS Internet Explorer and Mozilla Firefox. These two applications are covering over 90% of

market. Theoretically standards are created by W3C organisation
(https://www.w3.org/) but they are not always followed/implemented.

```
IE      FF      W3C  Omni completion ~
+/-     +/-     +    +               ~
+       +       -    +               ~
+       -       -    -               ~
-       +       -    -               ~
```

Regardless from state of implementation in browsers but if element is defined
in standards, completion plugin will place element in suggestion list. When
both major engines implemented element, even if this is not in standards it
will be suggested. All other elements are not placed in suggestion list.

LUA ft-lua-omni

Lua ftplugin sets 'omnifunc' to vim.lua_omnifunc().

PHP ft-php-omni

Completion of PHP code requires a tags file for completion of data from
external files and for class aware completion. You should use Universal/
Exuberant ctags version 5.5.4 or newer. You can find it here:

        Universal Ctags: https://ctags.io

Script completes:

- after $ variables name
    - if variable was declared as object add "->", if tags file is available

      show name of class
    - after "->" complete only function and variable names specific for given

      class. To find class location and contents tags file is required.
      Because PHP isn't strongly typed language user can use @var tag to declare
      class:

      ```
      /* @var $myVar myClass */
      $myVar->
      ```

      Still, to find myClass contents tags file is required.
- function names with additional info:
    - in case of built-in functions list of possible arguments and after | type

      data returned by function
    - in case of user function arguments and name of file where function was

      defined (if it is not current file)
- constants names
- class names after "new" declaration

Note: when doing completion first time Vim will load all necessary data into
memory. It may take several seconds. After next use of completion delay
should not be noticeable.

Script detects if cursor is inside <?php ?> tags. If it is outside it will
automatically switch to HTML/CSS/JavaScript completion. **Note:** contrary to
original HTML files completion of tags (and only tags) isn't context aware.

RUBY ft-ruby-omni

NOTE: compl-omni for Ruby code requires provider-ruby to be installed.

Ruby completion will parse your buffer on demand in order to provide a list of
completions. These completions will be drawn from modules loaded by "require"
and modules defined in the current buffer.

The completions provided by CTRL-X CTRL-O are sensitive to the context:

CONTEXT	COMPLETIONS PROVIDED
1. Not inside a class definition:	Classes, constants and globals
2. Inside a class definition:	Methods or constants defined in the class
3. After '.', '::' or ':':	Methods applicable to the object being dereferenced

4. After ':' or ':foo':   Symbol name (beginning with "foo")

Notes:

- Vim will load/evaluate code in order to provide completions. This may

  cause some code execution, which may be a concern. This is no longer
  enabled by default, to enable this feature add

```
let g:rubycomplete_buffer_loading = 1
```

- In context 1 above, Vim can parse the entire buffer to add a list of

  classes to the completion results. This feature is turned off by default,
  to enable it add

```
let g:rubycomplete_classes_in_global = 1
```

  to your vimrc
- In context 2 above, anonymous classes are not supported.
- In context 3 above, Vim will attempt to determine the methods supported by

  the object.
- Vim can detect and load the Rails environment for files within a rails

  project. The feature is disabled by default, to enable it add

```
let g:rubycomplete_rails = 1
```

  to your vimrc

SYNTAX **ft-syntax-omni**

Vim has the ability to color syntax highlight nearly 500 languages. Part of this highlighting includes knowing what keywords are part of a language. Many filetypes already have custom completion scripts written for them, the syntaxcomplete plugin provides basic completion for all other filetypes. It does this by populating the omni completion list with the text Vim already knows how to color highlight. It can be used for any filetype and provides a minimal language-sensitive completion.

To enable syntax code completion you can run:

```
setlocal omnifunc=syntaxcomplete#Complete
```

You can automate this by placing the following in your init.vim (after any ":filetype" command):

```
if has("autocmd") && exists("+omnifunc")
    autocmd Filetype *
                \       if &omnifunc == "" |
                \               setlocal omnifunc=syntaxcomplete#Complete |
                \       endif
endif
```

The above will set completion to this script only if a specific plugin does not already exist for that filetype.

Each filetype can have a wide range of syntax items. The plugin allows you to customize which syntax groups to include or exclude from the list. Let's have a look at the PHP filetype to see how this works.

If you edit a file called, index.php, run the following command:

```
syntax list
```

The first thing you will notice is that there are many different syntax groups. The PHP language can include elements from different languages like HTML, JavaScript and many more. The syntax plugin will only include syntax groups that begin with the filetype, "php", in this case. For example these syntax groups are included by default with the PHP: phpEnvVar, phpIntVar, phpFunctions.

If you wish non-filetype syntax items to also be included, you can use a regular expression syntax (added in version 13.0 of autoload/syntaxcomplete.vim) to add items. Looking at the output from ":syntax list" while editing a PHP file I can see some of these entries:

```
htmlArg,htmlTag,htmlTagName,javaScriptStatement,javaScriptGlobalObjects
```

To pick up any JavaScript and HTML keyword syntax groups while editing a PHP
file, you can use 3 different regexs, one for each language. Or you can
simply restrict the include groups to a particular value, without using
a regex string:

```
let g:omni_syntax_group_include_php = 'php\w\+,javaScript\w\+,html\w\+'
let g:omni_syntax_group_include_php = 'phpFunctions,phpMethods'
```

The basic form of this variable is:

```
let g:omni_syntax_group_include_{filetype} = 'regex,comma,separated'
```

The PHP language has an enormous number of items which it knows how to syntax
highlight. These items will be available within the omni completion list.

Some people may find this list unwieldy or are only interested in certain
items. There are two ways to prune this list (if necessary). If you find
certain syntax groups you do not wish displayed you can use two different
methods to identify these groups. The first specifically lists the syntax
groups by name. The second uses a regular expression to identify both
syntax groups. Simply add one the following to your vimrc:

```
let g:omni_syntax_group_exclude_php = 'phpCoreConstant,phpConstant'
let g:omni_syntax_group_exclude_php = 'php\w*Constant'
```

Add as many syntax groups to this list by comma separating them. The basic
form of this variable is:

```
let g:omni_syntax_group_exclude_{filetype} = 'regex,comma,separated'
```

You can create as many of these variables as you need, varying only the
filetype at the end of the variable name.

The plugin uses the isKeyword option to determine where word boundaries are
for the syntax items. For example, in the Scheme language completion should
include the "-", call-with-output-file. Depending on your filetype, this may
not provide the words you are expecting. Setting the
g:omni_syntax_use_iskeyword option to 0 will force the syntax plugin to break
on word characters. This can be controlled adding the following to your
vimrc:

```
let g:omni_syntax_use_iskeyword = 0
```

For plugin developers, the plugin exposes a public function
syntaxcomplete#OmniSyntaxList. This function can be used to request a List of
syntax items. When editing a SQL file (:e syntax.sql) you can use the
":syntax list" command to see the various groups and syntax items. For
example:

```
syntax list
```

Yields data similar to this:

```
sqlOperator    xxx some prior all like and any escape exists in is not
                   or intersect minus between distinct
                   links to Operator
sqlType        xxx varbit varchar nvarchar bigint int uniqueidentifier
                   date money long tinyint unsigned xml text smalldate
                   double datetime nchar smallint numeric time bit char
                   varbinary binary smallmoney
                   image float integer timestamp real decimal
```

There are two syntax groups listed here: sqlOperator and sqlType. To retrieve
a List of syntax items you can call syntaxcomplete#OmniSyntaxList a number of
different ways. To retrieve all syntax items regardless of syntax group:

```
echo syntaxcomplete#OmniSyntaxList( [] )
```

To retrieve only the syntax items for the sqlOperator syntax group:

```
echo syntaxcomplete#OmniSyntaxList( ['sqlOperator'] )
```

To retrieve all syntax items for both the sqlOperator and sqlType groups:

```
echo syntaxcomplete#OmniSyntaxList( ['sqlOperator', 'sqlType'] )
```

A regular expression can also be used:

```
echo syntaxcomplete#OmniSyntaxList( ['sql\w\+'] )
```

From within a plugin, you would typically assign the output to a List:

```
let myKeywords = []
let myKeywords = syntaxcomplete#OmniSyntaxList( ['sqlKeyword'] )
```

SQL **ft-sql-omni**

Completion for the SQL language includes statements, functions, keywords.
It will also dynamically complete tables, procedures, views and column lists
with data pulled directly from within a database. For detailed instructions
and a tutorial see omni-sql-completion.

The SQL completion plugin can be used in conjunction with other completion
plugins. For example, the PHP filetype has its own completion plugin.
Since PHP is often used to generate dynamic website by accessing a database,
the SQL completion plugin can also be enabled. This allows you to complete
PHP code and SQL code at the same time.

XML **ft-xml-omni**

Vim 7 provides a mechanism for context aware completion of XML files. It
depends on a special xml-omni-datafile and two commands: :XMLns and
:XMLent. Features are:

- after "<" complete the tag name, depending on context
- inside of a tag complete proper attributes
- when an attribute has a limited number of possible values help to complete them
- complete names of entities (defined in xml-omni-datafile and in the current file with "<!ENTITY" declarations)
- when used after "</" CTRL-X CTRL-O will close the last opened tag

Format of XML data file xml-omni-datafile

XML data files are stored in the "autoload/xml" directory in 'runtimepath'.
Vim distribution provides examples of data files in the
"$VIMRUNTIME/autoload/xml" directory. They have a meaningful name which will
be used in commands. It should be a unique name which will not create
conflicts. For example, the name xhtml10s.vim means it is the data file for
XHTML 1.0 Strict.

Each file contains a variable with a name like g:xmldata_xhtml10s . It is
a compound from two parts:

1. "g:xmldata_" general prefix, constant for all data files
2. "xhtml10s" the name of the file and the name of the described XML dialect; it will be used as an argument for the :XMLns command

Part two must be exactly the same as name of file.

The variable is a Dictionary. Keys are tag names and each value is a two
element List. The first element of the List is also a List with the names
of possible children. The second element is a Dictionary with the names of
attributes as keys and the possible values of attributes as values. Example:

```
let g:xmldata_crippled = {
\ "vimxmlentities": ["amp", "lt", "gt", "apos", "quot"],
\ 'vimxmlroot': ['tag1'],
\ 'tag1':
\ [ ['childoftag1a', 'childoftag1b'], {'attroftag1a': [],
\ 'attroftag1b': ['valueofattr1', 'valueofattr2']}],
\ 'childoftag1a':
\ [ [], {'attrofchild': ['attrofchild']}],
\ 'childoftag1b':
\ [ ['childoftag1a'], {'attrofchild': []}],
\ "vimxmltaginfo": {
\ 'tag1': ['Menu info', 'Long information visible in preview window']},
\ 'vimxmlattrinfo': {
\ 'attrofchild': ['Menu info', 'Long information visible in preview window']}}
```

This example would be put in the "autoload/xml/crippled.vim" file and could
help to write this file:

```
<tag1 attroftag1b="valueofattr1">
    <childoftag1a attrofchild>
            &amp;amp; &amp;lt;
    </childoftag1a>
    <childoftag1b attrofchild="5">
        <childoftag1a>
            &amp;gt; &amp;apos; &amp;quot;
        </childoftag1a>
    </childoftag1b>
</tag1>
```

In the example four special elements are visible:

1. "vimxmlentities" - a special key with List containing entities of this XML dialect.
2. If the list containing possible values of attributes has one element and this element is equal to the name of the attribute this attribute will be treated as boolean and inserted as "attrname" and not as 'attrname="'
3. "vimxmltaginfo" - a special key with a Dictionary containing tag names as keys and two element List as values, for additional menu info and the long description.
4. "vimxmlattrinfo" - special key with Dictionary containing attribute names as keys and two element List as values, for additional menu info and long description.

Note: Tag names in the data file MUST not contain a namespace description.
Check xsl.vim for an example.
Note: All data and functions are publicly available as global
variables/functions and can be used for personal editing functions.

DTD -> Vim dtd2vim

On www is the script dtd2vim which parses DTD and creates an XML data file
for Vim XML omni completion.

    dtd2vim: https://www.vim.org/scripts/script.php?script_id=1462

Check the beginning of that file for usage details.
The script requires perl and:

    perlSGML: https://savannah.nongnu.org/projects/perlsgml

Commands

:XMLns {name} [{namespace}] :XMLns

Vim has to know which data file should be used and with which namespace. For
loading of the data file and connecting data with the proper namespace use
:XMLns command. The first (obligatory) argument is the name of the data
(xhtml10s, xsl). The second argument is the code of namespace (h, xsl). When
used without a second argument the dialect will be used as default - without
namespace declaration. For example to use XML completion in .xsl files:

```
:XMLns xhtml10s
:XMLns xsl xsl
```

:XMLent {name} :XMLent

By default entities will be completed from the data file of the default
namespace. The XMLent command should be used in case when there is no default
namespace:

```
:XMLent xhtml10s
```

Usage

While used in this situation (after declarations from previous part, | is
cursor position):

```
<|
```

Will complete to an appropriate XHTML tag, and in this situation:

```
<xsl:|
```

Will complete to an appropriate XSL tag.

The script xmlcomplete.vim, provided through the autoload mechanism,
has the xmlcomplete#GetLastOpenTag() function which can be used in XML files
to get the name of the last open tag (b:unaryTagsStack has to be defined):

```
:echo xmlcomplete#GetLastOpenTag("b:unaryTagsStack")
```

8. Insert mode commands

inserting

The following commands can be used to insert new text into the buffer. They can all be undone and repeated with the "." command.

a

a Append text after the cursor [count] times. If the cursor is in the first column of an empty line Insert starts there. But not when 'virtualedit' is set!

A

A Append text at the end of the line [count] times. For using "A" in Visual block mode see v_b_A.

<insert> or i i insert <Insert>
Insert text before the cursor [count] times. When using CTRL-O in Insert mode i_CTRL-O the count is not supported.

I

I Insert text before the first non-blank in the line [count] times. For using "I" in Visual block mode see v_b_I.

gI

gI Insert text in column 1 [count] times.

gi

gi Insert text in the same position as where Insert mode was stopped last time in the current buffer. This uses the '^ mark. It's different from "`^i" when the mark is past the end of the line. The position is corrected for inserted/deleted lines, but NOT for inserted/deleted characters. When the :keepjumps command modifier is used the '^ mark won't be changed.

o

o Begin a new line below the cursor and insert text, repeat [count] times.

O

O Begin a new line above the cursor and insert text, repeat [count] times.

These commands are used to start inserting text. You can end insert mode with <Esc>. See mode-ins-repl for the other special characters in Insert mode. The effect of [count] takes place after Insert mode is exited.

The following default-mappings insert text, but stay in normal mode:

]<Space>

]<Space> Insert an empty line below the cursor without leaving Normal mode, repeat [count] times.

[<Space>

[<Space> Insert an empty line above the cursor without leaving Normal mode, repeat [count] times.

When 'autoindent' is on, the indent for a new line is obtained from the previous line. When 'smartindent' or 'cindent' is on, the indent for a line is automatically adjusted for C programs.

'formatoptions' can be set to copy the comment leader when opening a new line.

'textwidth' can be set to the maximum width for a line. When a line becomes too long when appending characters a line break is automatically inserted.

9. Ex insert commands

inserting-ex

:a :append
:{range}a[ppend][!] Insert several lines of text below the specified line. If the {range} is missing, the text will be inserted after the current line. Adding [!] toggles 'autoindent' for the time this command is executed.

:i :in :insert
:{range}i[nsert][!] Insert several lines of text above the specified line. If the {range} is missing, the text will be inserted before the current line. Adding [!] toggles 'autoindent' for the time this command is executed.

These two commands will keep on asking for lines, until you type a line containing only a ".". Watch out for lines starting with a backslash, see line-continuation.

Text typed after a "|" command separator is used first. So the following
command in Ex mode:

```
:a|one
two
.
:visual
```

appends the following text, after the cursor line:

```
one
two
```

NOTE: These commands cannot be used with :global or :vglobal.
":append" and ":insert" don't work properly in between ":if" and
":endif", ":for" and ":endfor", ":while" and ":endwhile".

:start :startinsert

:star[tinsert][!] Start Insert mode (or Terminal-mode in a terminal buffer) just after executing this command.
Works like typing "i" in Normal mode. When the ! is included it works like "A", append to the line.
Otherwise insertion starts at the cursor position.
Note that when using this command in a function or script, the insertion only starts after the function or script is finished.
This command does not work from :normal.

:stopi :stopinsert

:stopi[nsert] Stop Insert mode or Terminal-mode as soon as possible. Works like typing <Esc> in Insert mode.
Can be used in an autocommand, example:

```
:au BufEnter scratch stopinsert
```

replacing-ex :startreplace

:startr[eplace][!] Start Replace mode just after executing this command.
Works just like typing "R" in Normal mode. When the ! is included it acts just like "$R" had been typed (ie. begin replace mode at the end-of-line). Otherwise replacement begins at the cursor position.
Note that when using this command in a function or script that the replacement will only start after the function or script is finished.

:startgreplace

:startg[replace][!] Just like :startreplace, but use Virtual Replace mode, like with gR.

10. Inserting a file inserting-file

:r :re :read

:r[ead] [++opt] [name]
Insert the file [name] (default: current file) below the cursor.
See ++opt for the possible values of [++opt].

:{range}r[ead] [++opt] [name]
Insert the file [name] (default: current file) below the specified line.
See ++opt for the possible values of [++opt].

:r! :read!

:[range]r[ead] [++opt] !{cmd}
Execute {cmd} and insert its standard output below the cursor or the specified line. A temporary file is used to store the output of the command which is then read into the buffer. 'shellredir' is used to save the output of the command, which can be set to include stderr or not. {cmd} is executed like with ":!{cmd}", any '!' is replaced with the previous command :!.
See ++opt for the possible values of [++opt].

These commands insert the contents of a file, or the output of a command, into the buffer. They can be undone. They cannot be repeated with the "." command. They work on a line basis, insertion starts below the line in which the cursor is, or below the specified line. To insert text above the first line use the command ":0r {name}".

After the ":read" command, the cursor is left on the first non-blank in the first new line. If in Ex mode, then the cursor is left on the last new line (sorry, this is Vi compatible).

If a file name is given with ":r", it becomes the alternate file. This can be used, for example, when you want to edit that file instead: ":e! #". This can be switched off by removing the 'a' flag from the 'cpoptions' option.

Of the [++opt] arguments one is specifically for ":read", the ++edit argument.
This is useful when the ":read" command is actually used to read a file into the buffer as if editing that file. Use this command in an empty buffer:

```
:read ++edit filename
```

The effect is that the 'fileformat', 'fileencoding', 'bomb', etc. options are
set to what has been detected for "filename". Note that a single empty line
remains, you may want to delete it.

file-read

The 'fileformat' option sets the <EOL> style for a file:

'fileformat'	characters	name
"dos"	<CR><NL> or <NL>	DOS format
"unix"	<NL>	Unix format
"mac"	<CR>	Mac format

If 'fileformat' is "dos", a <CR> in front of an <NL> is ignored and a CTRL-Z
at the end of the file is ignored.

If 'fileformat' is "mac", a <NL> in the file is internally represented by a
<CR>. This is to avoid confusion with a <NL> which is used to represent a
<NUL>. See CR-used-for-NL.

If the 'fileformats' option is not empty Vim tries to recognize the type of
<EOL> (see file-formats). However, the 'fileformat' option will not be
changed, the detected format is only used while reading the file.
A similar thing happens with 'fileencodings'.

On non-Win32 systems the message "[dos format]" is shown if a file is read in
DOS format, to remind you that something unusual is done.
On Macintosh and Win32 the message "[unix format]" is shown if a file is read
in Unix format.
On non-Macintosh systems, the message "[mac format]" is shown if a file is
read in Mac format.

An example on how to use ":r !":

```
:r !uuencode binfile binfile
```

This command reads "binfile", uuencodes it and reads it into the current
buffer. Useful when you are editing e-mail and want to include a binary
file.

read-messages

When reading a file Vim will display a message with information about the read
file. In the table is an explanation for some of the items. The others are
self explanatory. Using the long or the short version depends on the
'shortmess' option.

long	short	meaning
[readonly]	{RO}	the file is write protected
[fifo/socket]	using a stream	
[fifo]	using a fifo stream	
[socket]	using a socket stream	
[CR missing]	reading with "dos" 'fileformat' and a NL without a preceding CR was found.	
[NL found]	reading with "mac" 'fileformat' and a NL was found (could be "unix" format)	
[long lines split]	at least one line was split in two	
[NOT converted]	conversion from 'fileencoding' to 'encoding' was desired but not possible	
[converted]	conversion from 'fileencoding' to 'encoding' done	
[READ ERRORS]	not all of the file could be read	

This file describes commands that delete or change text. In this context,
changing text means deleting the text and replacing it with other text using
one command. You can undo all of these commands. You can repeat the non-Ex
commands with the "." command.

For inserting text see insert.txt.

1. Deleting text

E470

["x]<Del>	or	<Del> x dl
["x]x		Delete [count] characters under and after the cursor
		[into register x] (not linewise). Does the same as
		"dl".
		The <Del> key does not take a [count]. Instead, it
		deletes the last character of the count.
		See 'whichwrap' for deleting a line break (join
		lines).

X dh
["x]X		Delete [count] characters before the cursor [into
		register x] (not linewise). Does the same as "dh".
		Also see 'whichwrap'.

d

["x]d{motion}		Delete text that {motion} moves over [into register
		x]. See below for exceptions.

dd

["x]dd		Delete [count] lines [into register x] linewise.

D

["x]D		Delete the characters under the cursor until the end
		of the line and [count]-1 more lines [into register
		x]; synonym for "d$".
		(not linewise)

{Visual}["x]x	or	v_x v_d v_<Del>
{Visual}["x]d	or
{Visual}["x]<Del>	Delete the highlighted text [into register x] (for
		{Visual} see Visual-mode).

{Visual}["x]CTRL-H	or	v_CTRL-H v_<BS>
{Visual}["x]<BS>	When in Select mode: Delete the highlighted text [into
		register x].

{Visual}["x]X	or	v_X v_D v_b_D
{Visual}["x]D		Delete the highlighted lines [into register x] (for
		{Visual} see Visual-mode). In Visual block mode,
		"D" deletes the highlighted text plus all text until
		the end of the line.

:d :de :del :delete :dl :dp
:[range]d[elete] [x]	Delete [range] lines (default: current line) [into
		register x].
		Note these weird abbreviations:
		  :dl		delete and list
		  :dell		idem
		  :delel	idem
		  :deletl	idem
		  :deletel	idem
		  :dp		delete and print
		  :dep		idem
		  :delp		idem
		  :delep	idem
		  :deletp	idem
		  :deletep	idem

:[range]d[elete] [x] {count}
		Delete {count} lines, starting with [range]
		(default: current line cmdline-ranges) [into
		register x].

These commands delete text. You can repeat them with the . command
(except :d) and undo them. Use Visual mode to delete blocks of text. See
registers for an explanation of registers.

d-special

An exception for the d{motion} command: If the motion is not linewise, the
start and end of the motion are not in the same line, and there are only
blanks before the start and there are no non-blanks after the end of the
motion, the delete becomes linewise. This means that the delete also removes
the line of blanks that you might expect to remain. Use the o_v operator to
force the motion to be charwise.

Trying to delete an empty region of text (e.g., "d0" in the first column)
is an error when 'cpoptions' includes the 'E' flag.

J

J		Join [count] lines, with a minimum of two lines.
		Remove the indent and insert up to two spaces (see
		below). Fails when on the last line of the buffer.

If [count] is too big it is reduced to the number of
lines available.

v_J

{Visual}J Join the highlighted lines, with a minimum of two
lines. Remove the indent and insert up to two spaces
(see below).

gJ

gJ Join [count] lines, with a minimum of two lines.
Don't insert or remove any spaces.

v_gJ

{Visual}gJ Join the highlighted lines, with a minimum of two
lines. Don't insert or remove any spaces.

:j :join

:[range]j[oin][!] [flags]
Join [range] lines. Same as "J", except with [!]
the join does not insert or delete any spaces.
If a [range] has equal start and end values, this
command does nothing. The default behavior is to
join the current line with the line below it.
See ex-flags for [flags].

:[range]j[oin][!] {count} [flags]
Join {count} lines, starting with [range] (default:
current line cmdline-ranges). Same as "J", except
with [!] the join does not insert or delete any
spaces.
See ex-flags for [flags].

These commands delete the <EOL> between lines. This has the effect of joining multiple lines into one line. You can repeat these commands (except :j) and undo them.

These commands, except "gJ", insert one space in place of the <EOL> unless there is trailing white space or the next line starts with a ')'. These commands, except "gJ", delete any leading white space on the next line. If the 'joinspaces' option is on, these commands insert two spaces after a '.', '!' or '?'.
The 'B' and 'M' flags in 'formatoptions' change the behavior for inserting spaces before and after a multibyte character fo-table.

The '[mark is set at the end of the first line that was joined, '] at the end of the resulting line.

2. Delete and insert delete-insert replacing

R

R Enter Replace mode: Each character you type replaces
an existing character, starting with the character
under the cursor. Repeat the entered text [count]-1
times. See Replace-mode for more details.

gR

gR Enter Virtual Replace mode: Each character you type
replaces existing characters in screen space. So a
<Tab> may replace several characters at once.
Repeat the entered text [count]-1 times. See
Virtual-Replace-mode for more details.

c

["x]c{motion} Delete {motion} text [into register x] and start
insert. When 'cpoptions' includes the 'E' flag and
there is no text to delete (e.g., with "cTx" when the
cursor is just after an 'x'), an error occurs and
insert mode does not start (this is Vi compatible).
When 'cpoptions' does not include the 'E' flag, the
"c" command always starts insert mode, even if there
is no text to delete.

cc

["x]cc Delete [count] lines [into register x] and start
insert linewise. If 'autoindent' is on, preserve
the indent of the first line.

C

["x]C Delete from the cursor position to the end of the
line and [count]-1 more lines [into register x], and
start insert. Synonym for c$ (not linewise).

s

["x]s Delete [count] characters [into register x] and start
insert (s stands for Substitute). Synonym for "cl"
(not linewise).

S

["x]S Delete [count] lines [into register x] and start insert. Synonym for "cc" linewise.

{Visual}["x]c or {Visual}["x]s — v_c v_s
Delete the highlighted text [into register x] and start insert (for {Visual} see Visual-mode).

v_r

{Visual}r{char} Replace all selected characters by {char}. CTRL-C will be inserted literally.

v_C

{Visual}["x]C Delete the highlighted lines [into register x] and start insert. In Visual block mode it works differently v_b_C.

v_S

{Visual}["x]S Delete the highlighted lines [into register x] and start insert (for {Visual} see Visual-mode).

v_R

{Visual}["x]R Currently just like {Visual}["x]S. In a next version it might work differently.

Notes:

- You can end Insert and Replace mode with <Esc>.
- See the section "Insert and Replace mode" mode-ins-repl for the other special characters in these modes.
- The effect of [count] takes place after Vim exits Insert or Replace mode.
- When the 'cpoptions' option contains '$' and the change is within one line, Vim continues to show the text to be deleted and puts a '$' at the last deleted character.

See registers for an explanation of registers.

Replace mode is just like Insert mode, except that every character you enter deletes one character. If you reach the end of a line, Vim appends any further characters (just like Insert mode). In Replace mode, the backspace key restores the original text (if there was any). (See section "Insert and Replace mode" mode-ins-repl).

cw cW

Special case: When the cursor is in a word, "cw" and "cW" do not include the white space after a word, they only change up to the end of the word. This is because Vim interprets "cw" as change-word, and a word does not include the following white space.

If you prefer "cw" to include the space after a word, see cpo-_ to change the behavior. Alternatively, use "caw" (see aw).

:c :ch :change

:{range}c[hange][!] Replace lines of text with some different text. Type a line containing only "." to stop replacing. Without {range}, this command changes only the current line.
Adding [!] toggles 'autoindent' for the time this command is executed.

3. Simple changes simple-change changing

r

r{char} Replace the character under the cursor with {char}. If {char} is a <CR> or <NL>, a line break replaces the character. To replace with a real <CR>, use CTRL-V <CR>. CTRL-V <NL> replaces with a <Nul>.

If {char} is CTRL-E or CTRL-Y the character from the line below or above is used, just like with i_CTRL-E and i_CTRL-Y. This also works with a count, thus 10r<C-E> copies 10 characters from the line below.

If you give a [count], Vim replaces [count] characters with [count] {char}s. When {char} is a <CR> or <NL>, however, Vim inserts only one <CR>: "5r<CR>" replaces five characters with a single line break.
When {char} is a <CR> or <NL>, Vim performs autoindenting. This works just like deleting the characters that are replaced and then doing "i<CR><Esc>".
{char} can be entered as a digraph digraph-arg.
:lmap mappings apply to {char}. The CTRL-^ command in Insert mode can be used to switch this on/off

i_CTRL-^. See utf-8-char-arg about using composing characters when 'encoding' is Unicode.

gr

gr{char} Replace the virtual characters under the cursor with {char}. This replaces in screen space, not file space. See gR and Virtual-Replace-mode for more details. As with r a count may be given. {char} can be entered like with r, but characters that have a special meaning in Insert mode, such as most CTRL-keys, cannot be used.

gr-default

Nvim creates lsp-defaults mappings which may inhibit the builtin behavior of gr. Use this to restore the builtin behavior:

```
nnoremap <nowait> gr gr
```

digraph-arg

The argument for Normal mode commands like r and t is a single character. When 'cpo' doesn't contain the 'D' flag, this character can also be entered like digraphs. First type CTRL-K and then the two digraph characters.

case

The following commands change the case of letters. The currently active locale is used. See :language. The LC_CTYPE value matters here.

~

~ 'notildeop' option: Switch case of the character under the cursor and move the cursor to the right. If a [count] is given, do that many characters.

~{motion} 'tildeop' option: switch case of {motion} text.

g~

g~{motion} Switch case of {motion} text.

g~g~ g~g~ g~~

g~~ Switch case of current line.

v_~

{Visual}~ Switch case of highlighted text (for {Visual} see Visual-mode).

v_U

{Visual}U Make highlighted text uppercase (for {Visual} see Visual-mode).

gU uppercase

gU{motion} Make {motion} text uppercase. Example:

```
:map! <C-F> <Esc>gUiw`]a
```

This works in Insert mode: press CTRL-F to make the word before the cursor uppercase. Handy to type words in lowercase and then make them uppercase.

gUgU gUgU gUU

gUU Make current line uppercase.

v_u

{Visual}u Make highlighted text lowercase (for {Visual} see Visual-mode).

gu lowercase

gu{motion} Make {motion} text lowercase.

gugu gugu guu

guu Make current line lowercase.

g? rot13

g?{motion} Rot13 encode {motion} text.

v_g?

{Visual}g? Rot13 encode the highlighted text (for {Visual} see Visual-mode).

g?g? g?g? g??

g?? Rot13 encode current line.

To turn one line into title caps, make every first letter of a word uppercase:

```
:s/\v<(.)(\w*)/\u\1\L\2/g
```

Adding and subtracting

CTRL-A

CTRL-A Add [count] to the number or alphabetic character at or after the cursor.

v_CTRL-A

{Visual}CTRL-A Add [count] to the number or alphabetic character in the highlighted text.

v_g_CTRL-A

{Visual}g CTRL-A Add [count] to the number or alphabetic character in the highlighted text. If several lines are highlighted, each one will be incremented by an additional [count] (so effectively creating a [count] incrementing sequence).
For Example, if you have this list of numbers:

```
1.
1.
1.
1.
```

Move to the second "1." and Visually select three lines, pressing g CTRL-A results in:

```
1.
2.
3.
4.
```

CTRL-X

CTRL-X Subtract [count] from the number or alphabetic character at or after the cursor.

v_CTRL-X

{Visual}CTRL-X Subtract [count] from the number or alphabetic character in the highlighted text.

v_g_CTRL-X

{Visual}g CTRL-X Subtract [count] from the number or alphabetic character in the highlighted text. If several lines are highlighted, each value will be decremented by an additional [count] (so effectively creating a [count] decrementing sequence).

The CTRL-A and CTRL-X commands work for (signed) decimal numbers, unsigned binary/octal/hexadecimal numbers and alphabetic characters.

This depends on the 'nrformats' option:

- When 'nrformats' includes "bin", Vim assumes numbers starting with '0b' or '0B' are binary.
- When 'nrformats' includes "octal", Vim considers numbers starting with a '0' to be octal, unless the number includes a '8' or '9'. Other numbers are decimal and may have a preceding minus sign.
  If the cursor is on a number, the commands apply to that number; otherwise Vim uses the number to the right of the cursor.
- When 'nrformats' includes "hex", Vim assumes numbers starting with '0x' or '0X' are hexadecimal. The case of the rightmost letter in the number determines the case of the resulting hexadecimal number. If there is no letter in the current number, Vim uses the previously detected case.
- When 'nrformats' includes "alpha", Vim will change the alphabetic character under or after the cursor. This is useful to make lists with an alphabetic index.

For decimals a leading negative sign is considered for incrementing or decrementing, for binary, octal and hex values, it won't be considered. To ignore the sign Visually select the number before using CTRL-A or CTRL-X.

For numbers with leading zeros (including all octal and hexadecimal numbers), Vim preserves the number of characters in the number when possible. CTRL-A on "0077" results in "0100", CTRL-X on "0x100" results in "0x0ff".
There is one exception: When a number that starts with a zero is found not to be octal (it contains a '8' or '9'), but 'nrformats' does include "octal", leading zeros are removed to avoid that the result may be recognized as an octal number.

Note that when 'nrformats' includes "octal", decimal numbers with leading zeros cause mistakes, because they can be confused with octal numbers.

Note similarly, when 'nrformats' includes both "bin" and "hex", binary numbers with a leading '0x' or '0X' can be interpreted as hexadecimal rather than binary since '0b' are valid hexadecimal digits. CTRL-A on "0x0b11" results in

"0x0b12", not "0x0b100".
When 'nrformats' includes "bin" and doesn't include "hex", CTRL-A on "0b11" in
"0x0b11" results in "0x0b100".

When the number under the cursor is too big to fit into 64 bits, it will be
rounded off to the nearest number that can be represented, and the
addition/subtraction is skipped. E.g. CTRL-X on 18446744073709551616 results
in 18446744073709551615. Same for larger numbers, such as 18446744073709551618.

The CTRL-A command is very useful in a macro. Example: Use the following
steps to make a numbered list.

1. Create the first list entry, make sure it starts with a number.

2. qa   - start recording into register 'a'

3. Y   - yank the entry

4. p   - put a copy of the entry below the first one

5. CTRL-A   - increment the number

6. q   - stop recording

7. <count>@a - repeat the yank, put and increment <count> times

SHIFTING LINES LEFT OR RIGHT **shift-left-right**

<

<{motion} Shift {motion} lines one 'shiftwidth' leftwards.

If the 'shiftwidth' option is set to zero, the amount
of indent is calculated at the first non-blank
character in the line.

<<

<< Shift [count] lines one 'shiftwidth' leftwards.

v_<

{Visual}[count]< Shift the highlighted lines [count] 'shiftwidth'
leftwards (for {Visual} see Visual-mode).

>

>{motion} Shift {motion} lines one 'shiftwidth' rightwards.

If the 'shiftwidth' option is set to zero, the amount
of indent is calculated at the first non-blank
character in the line.

>>

>> Shift [count] lines one 'shiftwidth' rightwards.

v_>

{Visual}[count]> Shift the highlighted lines [count] 'shiftwidth'
rightwards (for {Visual} see Visual-mode).

:<

:[range]< Shift [range] lines one 'shiftwidth' left. Repeat '<'
for shifting multiple 'shiftwidth's.

:[range]< {count} Shift {count} lines one 'shiftwidth' left, starting
with [range] (default current line cmdline-ranges).
Repeat '<' for shifting multiple 'shiftwidth's.

:[range]le[ft] [indent] left align lines in [range]. Sets the indent in the
lines to [indent] (default 0).

:>

:[range]> [flags] Shift [range] lines one 'shiftwidth' right.
Repeat '>' for shifting multiple 'shiftwidth's.
See ex-flags for [flags].

:[range]> {count} [flags]
Shift {count} lines one 'shiftwidth' right, starting
with [range] (default current line cmdline-ranges).
Repeat '>' for shifting multiple 'shiftwidth's.
See ex-flags for [flags].

The ">" and "<" commands are handy for changing the indentation within
programs. Use the 'shiftwidth' option to set the size of the white space
which these commands insert or delete. Normally the 'shiftwidth' option is 8,
but you can set it to, say, 3 to make smaller indents. The shift leftwards
stops when there is no indent. The shift right does not affect empty lines.

If the 'shiftround' option is on, the indent is rounded to a multiple of 'shiftwidth'.

If the 'smartindent' option is on, or 'cindent' is on and 'cinkeys' contains '#' with a zero value, shift right does not affect lines starting with '#' (these are supposed to be C preprocessor lines that must stay in column 1). This can be changed with the 'cino' option, see cino-#.

When the 'expandtab' option is off (this is the default) Vim uses <Tab>s as much as possible to make the indent. You can use ">><<" to replace an indent made out of spaces with the same indent made out of <Tab>s (and a few spaces if necessary). If the 'expandtab' option is on, Vim uses only spaces. Then you can use ">><<" to replace <Tab>s in the indent by spaces (or use :retab!).

To move a line several 'shiftwidth's, use Visual mode or the : commands. For example:

```
Vjj4>           move three lines 4 indents to the right
:<<<            move current line 3 indents to the left
:>> 5           move 5 lines 2 indents to the right
:5>>            move line 5 2 indents to the right
```

4. Complex changes complex-change

4.1 Filter commands filter

A filter is a program that accepts text at standard input, changes it in some way, and sends it to standard output. You can use the commands below to send some text through a filter, so that it is replaced by the filter output. Examples of filters are "sort", which sorts lines alphabetically, and "indent", which formats C program files (you need a version of indent that works like a filter; not all versions do). The 'shell' option specifies the shell Vim uses to execute the filter command. You can repeat filter commands with ".". Vim does not recognize a comment (starting with '"') after the :! command.

!

!{motion}{filter} Filter {motion} text lines through the external program {filter}.

!!

!!{filter} Filter [count] lines through the external program {filter}.

v_!

{Visual}!{filter} Filter the highlighted lines through the external program {filter} (for {Visual} see Visual-mode).

:{range}![!]{filter} [!][arg] :range!

For executing external commands see :!

Filter {range} lines through the external program {filter}. Vim replaces the optional bangs with the latest given command and appends the optional [arg]. Vim saves the output of the filter command in a temporary file and then reads the file into the buffer tempfile. Vim uses the 'shellredir' option to redirect the filter output to the temporary file. However, if the 'shelltemp' option is off then pipes are used when possible (on Unix). When the 'R' flag is included in 'cpoptions' marks in the filtered lines are deleted, unless the :keepmarks command is used. Example:

```
:keepmarks '<,'>!sort
```

When the number of lines after filtering is less than before, marks in the missing lines are deleted anyway.

=

={motion} Filter {motion} lines through the external program given with the 'equalprg' option. When the 'equalprg' option is empty (this is the default), use the internal formatting function C-indenting and 'lisp'. But when 'indentexpr' is not empty, it will be used instead indent-expression.

==

== Filter [count] lines like with ={motion}.

v_=

{Visual}= Filter the highlighted lines like with ={motion}.

tempdir tempfile setuid

Nvim uses temporary files for filtering and generating diffs. Plugins also commonly use tempname() for their own purposes. On the first request for a temporary file, Nvim creates a common directory (the "Nvim tempdir"), to

serve as storage for all temporary files (including stdpath("run") files
$XDG_RUNTIME_DIR) in the current session.

The Nvim tempdir is created in the first available system tempdir:
        Unix: $TMPDIR, /tmp, current-dir, $HOME.
        Windows: $TMPDIR, $TMP, $TEMP, $USERPROFILE, current-dir.

On unix the tempdir is created with permissions 0700 (only accessible by the
current user) to avoid security problems (e.g. symlink attacks). On exit,
Nvim deletes the tempdir and its contents.

E5431

If you see an error or log message like:

```
E5431: tempdir disappeared (2 times)
```

this means an external process on your system deleted the Nvim tempdir.
Typically this is caused by "antivirus" or a misconfigured cleanup service.

If Nvim has the setuid bit set this may cause problems: the temp file
is owned by the setuid user but the filter command probably runs as the
original user.

4.2 Substitute :substitute

:s :su
:[range]s[ubstitute]/{pattern}/{string}/[flags] [count]
                        For each line in [range] replace a match of {pattern}
                        with {string}.
                        For the {pattern} see pattern.
                        {string} can be a literal string, or something
                        special; see sub-replace-special.
                        When [range] and [count] are omitted, replace in the
                        current line only. When [count] is given, replace in
                        [count] lines, starting with the last line in [range].
                        When [range] is omitted start in the current line.
                                                E939 E1510
                        [count] must be a positive number (max 2147483647)
                        Also see cmdline-ranges.

                        See :s_flags for [flags].
                        The delimiter doesn't need to be /, see
                        pattern-delimiter.

:[range]s[ubstitute] [flags] [count]
:[range]&[&][flags] [count]   :&

                        Repeat last :substitute with same search pattern and
                        substitute string, but without the same flags. You
                        may add [flags], see :s_flags.
                        Note that after :substitute the '&' and '#' flags
                        can't be used, they're recognized as a pattern
                        separator.
                        The space between :substitute and the 'c', 'g',
                        'i', 'I' and 'r' flags isn't required, but in scripts
                        it's a good idea to keep it to avoid confusion.
                        Also see the two and three letter commands to repeat
                        :substitute below :substitute-repeat.

:[range]~[&][flags] [count]   :~

                        Repeat last substitute with same substitute string
                        but with last used search pattern. This is like
                        :&r. See :s_flags for [flags].

                                                                        &

&   Synonym for :s (repeat last substitute). Note
                        that the flags are not remembered, thus it might
                        actually work differently. You can use :&& to keep
                        the flags.

                                                                        &-default

                        Mapped to ":&&<CR>" by default. default-mappings

                                                                        g&

g&   Synonym for :%s//~/& (repeat last substitute with
                        last search pattern on all lines with the same flags).
                        For example, when you first do a substitution with
                        :s/pattern/repl/flags and then /search for
                        something else, g& will do :%s/search/repl/flags.
                        Mnemonic: global substitute.

                                                :snomagic :sno
:[range]sno[magic] ... Same as :substitute, but always use 'nomagic'.

                                                :smagic :sm
:[range]sm[agic] ...   Same as :substitute, but always use 'magic'.

                                                                        :s_flags

The flags that you can use for the substitute commands:

                                                                        :&&

[&]	Must be the first one: Keep the flags from the previous substitute
	command. Examples:

```
:&&
:s/this/that/&
```

	Note that :s and :& don't keep the flags.

[c]	Confirm each substitution. Vim highlights the matching string (with
	hl-IncSearch). You can type: :s_c

```
    'y'     to substitute this match
    'l'     to substitute this match and then quit ("last")
    'n'     to skip this match
    <Esc>   to quit substituting
    'a'     to substitute this and all remaining matches
    'q'     to quit substituting
    CTRL-E  to scroll the screen up
    CTRL-Y  to scroll the screen down
```

:s_e

[e]	When the search pattern fails, do not issue an error message and, in
	particular, continue in maps as if no error occurred. This is most
	useful to prevent the "No match" error from breaking a mapping. Vim
	does not suppress the following error messages, however:
		Regular expressions can't be delimited by letters
		\ should be followed by /, ? or &
		No previous substitute regular expression
		Trailing characters
		Interrupted

:s_g

[g]	Replace all occurrences in the line. Without this argument,
	replacement occurs only for the first occurrence in each line. If the
	'gdefault' option is on, this flag is on by default and the [g]
	argument switches it off.

:s_i

[i]	Ignore case for the pattern. The 'ignorecase' and 'smartcase' options
	are not used.

:s_I

[I]	Don't ignore case for the pattern. The 'ignorecase' and 'smartcase'
	options are not used.

:s_n

[n]	Report the number of matches, do not actually substitute. The [c]
	flag is ignored. The matches are reported as if 'report' is zero.
	Useful to count-items.
	If \= sub-replace-expression is used, the expression will be
	evaluated in the sandbox at every match.

[p]	Print the line containing the last substitute. :s_p

[#]	Like [p] and prepend the line number. :s_#

[l]	Like [p] but print the text like :list. :s_l

:s_r

[r]	Only useful in combination with :& or :s without arguments. :&r
	works the same way as :~: When the search pattern is empty, use the
	previously used search pattern instead of the search pattern from the
	last substitute or :global. If the last command that did a search
	was a substitute or :global, there is no effect. If the last
	command was a search command such as "/", use the pattern from that
	command.
	For :s with an argument this already happens:

```
:s/blue/red/
/green
:s//red/   or  :~   or  :&r
```

	The last commands will replace "green" with "red".

```
:s/blue/red/
/green
:&
```

	The last command will replace "blue" with "red".

Note that there is no flag to change the "magicness" of the pattern. A
different command is used instead, or you can use /\v and friends. The
reason is that the flags can only be found by skipping the pattern, and in
order to skip the pattern the "magicness" must be known. Catch 22!

If the {pattern} for the substitute command is empty, the command uses the
pattern from the last substitute or :global command. If there is none, but
there is a previous search pattern, that one is used. With the [r] flag, the
command uses the pattern from the last substitute, :global, or search
command.

If the {string} is omitted the substitute is done as if it's empty. Thus the matched pattern is deleted. The separator after {pattern} can also be left out then. Example:

```
:%s/TESTING
```

This deletes "TESTING" from all lines, but only one per line.

For compatibility with Vi these two exceptions are allowed:
"\/{string}/" and "\?{string}?" do the same as "//{string}/r".
"\&{string}&" does the same as "//{string}/".

pattern-delimiter E146

Instead of the '/' which surrounds the pattern and replacement string, you can use another single-byte character. This is useful if you want to include a '/' in the search pattern or replacement string. Example:

```
:s+/+//+
```

You can use most characters, but not an alphanumeric character, '\', '"' or '|'.

For the definition of a pattern, see pattern. In Visual block mode, use /\%V in the pattern to have the substitute work in the block only. Otherwise it works on whole lines anyway.

sub-replace-special :s\=

When the {string} starts with "\=" it is evaluated as an expression, see sub-replace-expression. You can use that for complex replacement or special characters.

The substitution is limited in recursion to 4 levels. E1290

Otherwise these characters in {string} have a special meaning:

magic	nomagic	action
&	\&	replaced with the whole matched pattern s/\&
\&	&	replaced with &
\0	replaced with the whole matched pattern	\0 s/\0
\1	replaced with the matched pattern in the first	
pair of ()	s/\1	
\2	replaced with the matched pattern in the second	
pair of ()	s/\2	
..	..	s/\3
\9	replaced with the matched pattern in the ninth	
pair of ()	s/\9	
~	\~	replaced with the {string} of the previous
substitute	s~	
\~	~	replaced with ~ s/\~
\u	next character made uppercase	s/\u
\U	following characters made uppercase, until \E s/\U	
\l	next character made lowercase	s/\l
\L	following characters made lowercase, until \E s/\L	
\e	end of \u, \U, \l and \L (**NOTE:** not <Esc>!)	s/\e
\E	end of \u, \U, \l and \L	s/\E
<CR>	split line in two at this point	
(Type the <CR> as CTRL-V <Enter>)	s<CR>	
\r	idem	s/\r
\<CR>	insert a carriage-return (CTRL-M)	
(Type the <CR> as CTRL-V <Enter>)	s/\<CR>	
\n	insert a <NL> (<NUL> in the file)	
(does NOT break the line)	s/\n	
\b	insert a <BS>	s/\b
\t	insert a <Tab>	s/\t
\\	insert a single backslash	s/\\
\x	where x is any character not mentioned above: Reserved for future expansion	

The special meaning is also used inside the third argument {sub} of the substitute() function with the following exceptions:

- A % inserts a percent literally without regard to 'cpoptions'.

- magic is always set without regard to 'magic'.

- A ~ inserts a tilde literally.

- <CR> and \r inserts a carriage-return (CTRL-M).

- \<CR> does not have a special meaning. It's just one of \x.

Examples:

```
:s/a\|b/xxx\0xxx/g              modifies "a b"      to "xxxaxxx xxxbxxx"
:s/\([abc]\)\([efg]\)/\2\1/g    modifies "af fa bg" to "fa fa gb"
:s/abcde/abc^Mde/               modifies "abcde"    to "abc", "de" (two lines)
:s/$/\^M/                       modifies "abcde"    to "abcde^M"
:s/\w\+/\u\0/g          modifies "bla bla"  to "Bla Bla"
:s/\w\+/\L\u\0/g                modifies "BLA bla"  to "Bla Bla"
```

Note: "\L\u" can be used to capitalize the first letter of a word. This is not compatible with Vi and older versions of Vim, where the "\u" would cancel out the "\L". Same for "\U\l".

Note: In previous versions CTRL-V was handled in a special way. Since this is not Vi compatible, this was removed. Use a backslash instead.

command	text	result
:s/aa/a^Ma/	aa	a<line-break>a
:s/aa/a\^Ma/	aa	a^Ma
:s/aa/a\\^Ma/	aa	a\<line-break>a

(you need to type CTRL-V <CR> to get a ^M here)

The numbering of "\1", "\2" etc. is done based on which "\(" comes first in the pattern (going left to right). When a parentheses group matches several times, the last one will be used for "\1", "\2", etc. Example:

```
:s/\(\(a[a-d] \)*\)/\2/      modifies "aa ab x" to "ab x"
```

The "\2" is for "\(a[a-d] \)". At first it matches "aa ", secondly "ab ".

When using parentheses in combination with '|', like in \([ab]\)\|\([cd]\), either the first or second pattern in parentheses did not match, so either \1 or \2 is empty. Example:

```
:s/\([ab]\)\|\([cd]\)/\1x/g   modifies "a b c d"  to "ax bx x x"
```

:sc :sce :scg :sci :scI :scl :scp :sg :sgc
:sge :sgi :sgI :sgl :sgn :sgp :sgr :sI :si
:sic :sIc :sie :sIe :sIg :sIl :sin :sIn :sIp
:sip :sIr :sir :sr :src :srg :sri :srI :srl
:srn :srp :substitute-repeat

2-letter and 3-letter :substitute commands

These commands repeat the previous :substitute command with the given flags. The first letter is always "s", followed by one or two of the possible flag characters. For example ":sce" works like ":s ce". The table lists the possible combinations, not all flags are possible, because the command is short for another command.

```
    List of :substitute commands
    |      c    e    g    i    I    n    p    l    r
    | c  :sc  :sce :scg :sci :scI :scn :scp :scl
    | e
    | g  :sgc :sge :sg  :sgi :sgI :sgn :sgp :sgl :sgr
    | i  :sic :sie      :si  :siI :sin :sip      :sir
    | I  :sIc :sIe :sIg :sIi :sI  :sIn :sIp :sIl :sIr
    | n
    | p
    | l
    | r  :src      :srg :sri :srI :srn :srp :srl :sr
```

Exceptions:

```
    :scr  is  :scriptnames
    :se   is  :set
    :sig  is  :sign
    :sil  is  :silent
    :sn   is  :snext
    :sp   is  :split
    :sl   is  :sleep
    :sre  is  :srewind
```

Substitute with an expression sub-replace-expression

sub-replace-\= s/\=

When the substitute string starts with "\=" the remainder is interpreted as an expression.

The special meaning for characters as mentioned at sub-replace-special does not apply except for "<CR>". A <NL> character is used as a line break, you can get one with a double-quote string: "\n". Prepend a backslash to get a real <NL> character (which will be a NUL in the file).

The "\=" notation can also be used inside the third argument {sub} of substitute() function. In this case, the special meaning for characters as mentioned at sub-replace-special does not apply at all. Especially, <CR> and <NL> are interpreted not as a line break but as a carriage-return and a new-line respectively.

When the result is a List then the items are joined with separating line breaks. Thus each item becomes a line, except that they can contain line breaks themselves.

The submatch() function can be used to obtain matched text. The whole matched text can be accessed with "submatch(0)". The text matched with the first pair of () with "submatch(1)". Likewise for further sub-matches in ().

Be careful: The separation character must not appear in the expression! Consider using a character like "@" or ":". There is no problem if the result of the expression contains the separation character.

Examples:

```
:s@\n@\="\r" .. expand("$HOME") .. "\r"@
```

This replaces an end-of-line with a new line containing the value of $HOME.

```
s/E/\="\<Char-0x20ac>"/g
```

This replaces each 'E' character with a euro sign. Read more in <Char->.

4.3 Changing tabs change-tabs

:ret :retab :retab!

:[range]ret[ab][!] [-indentonly] [{new-tabstop}]
Replace all sequences of white-space containing a <Tab> with new strings of white-space using {new-tabstop}. If you do not specify {new-tabstop} or it is zero, Vim uses the current value of 'tabstop'. The current value of 'tabstop' is always used to compute the width of existing tabs.
With !, Vim also replaces strings of only normal spaces with tabs where appropriate.
With 'expandtab' on, Vim replaces all tabs with the appropriate number of spaces.
This command sets 'tabstop' to {new-tabstop} and if performed on the whole file, which is default, should not make any visible change.

When [-indentonly] is specified, only the leading white-space will be targeted. Any other consecutive white-space will not be changed.

Warning: This command modifies any <Tab> characters inside of strings in a C program. Use "\t" to avoid this (that's a good habit anyway).
:retab! may also change a sequence of spaces by <Tab> characters, which can mess up a printf().
A list of tab widths separated by commas may be used in place of a single tabstop. Each value in the list represents the width of one tabstop, except the final value which applies to all following tabstops.

retab-example

Example for using autocommands and ":retab" to edit a file which is stored with tabstops at 8 but edited with tabstops set at 4. **Warning:** white space inside of strings can change! Also see 'softtabstop' option.

```
:auto BufReadPost       *.xx    retab! 4
:auto BufWritePre       *.xx    retab! 8
:auto BufWritePost      *.xx    retab! 4
:auto BufNewFile        *.xx    set ts=4
```

5. Copying and moving text copy-move

quote

"{register} Use {register} for next delete, yank or put. Use an uppercase character to append with delete and yank. Registers ".", "%", "#" and ":" only work with put.

:reg :registers

:reg[isters] Display the type and contents of all numbered and named registers. If a register is written to for :redir it will not be listed.
Type can be one of:
"c" for characterwise text

"l"	for linewise text
"b"	for blockwise-visual text

:reg[isters] {arg}	Display the contents of the numbered and named registers that are mentioned in {arg}. For example:

```
:reg 1a
```

to display registers '1' and 'a'. Spaces are allowed in {arg}.

:di :dis :display

:di[splay] [arg]	Same as :registers.

y yank

["x]y{motion}	Yank {motion} text [into register x]. When no characters are to be yanked (e.g., "y0" in column 1), this is an error when 'cpoptions' includes the 'E' flag.

yy

["x]yy	Yank [count] lines [into register x] linewise.

Y

["x]Y	yank [count] lines [into register x] (synonym for yy, linewise).

Y-default

Mapped to "y$" by default. default-mappings

zy

["x]zy{motion}	Yank {motion} text [into register x]. Only differs from y when selecting a block of text, see v_zy.

v_y

{Visual}["x]y	Yank the highlighted text [into register x] (for {Visual} see Visual-mode).

v_Y

{Visual}["x]Y	Yank the highlighted lines [into register x] (for {Visual} see Visual-mode).

v_zy

{Visual}["x]zy	Yank the highlighted text [into register x]. Trailing whitespace at the end of each line of a selected block won't be yanked. Especially useful in combination with zp. (for {Visual} see Visual-mode)

:y :yank E850

:[range]y[ank] [x]	Yank [range] lines [into register x].

:[range]y[ank] [x] {count}
Yank {count} lines, starting with last line number in [range] (default: current line cmdline-ranges), [into register x].

p put E353 E1240

["x]p	Put the text [from register x] after the cursor [count] times.

P

["x]P	Put the text [from register x] before the cursor [count] times.

<MiddleMouse>

["x]<MiddleMouse>	Put the text from a register before the cursor [count] times. Uses the "* register, unless another is specified.
Leaves the cursor at the end of the new text.
Using the mouse only works when 'mouse' contains 'n' or 'a'.
If you have a scrollwheel and often accidentally paste text, you can use these mappings to disable the pasting with the middle mouse button:

```
:map <MiddleMouse> <Nop>
:imap <MiddleMouse> <Nop>
```

You might want to disable the multi-click versions too, see double-click.

gp

["x]gp	Just like "p", but leave the cursor just after the new text.

gP

["x]gP	Just like "P", but leave the cursor just after the new text.

:pu :put

:[line]pu[t] [x] Put the text [from register x] after [line] (default current line). This always works linewise, thus this command can be used to put a yanked block as new lines.
If no register is specified, it depends on the 'clipboard' option: If 'clipboard' contains "unnamedplus", paste from the + register quoteplus. Otherwise, if 'clipboard' contains "unnamed", paste from the * register quotestar. Otherwise, paste from the unnamed register quote_quote.
The register can also be '=' followed by an optional expression. The expression continues until the end of the command. You need to escape the '|' and '"' characters to prevent them from terminating the command. Example:

```
:put ='path' .. \",/test\"
```

If there is no expression after '=', Vim uses the previous expression. You can see it with ":dis =".

:[line]pu[t]! [x] Put the text [from register x] before [line] (default current line).

:ip :iput

:[line]ip[ut] [x] Like :put, but adjust indent to the current line.

:[line]ip[ut]! [x] Like :put!, but adjust indent to the current line.

["x]]p or]p]<MiddleMouse>
["x]]<MiddleMouse> Like "p", but adjust the indent to the current line. Using the mouse only works when 'mouse' contains 'n' or 'a'.

["x][P or [P

["x]]P or]P

["x][p or [p [<MiddleMouse>
["x][<MiddleMouse> Like "P", but adjust the indent to the current line. Using the mouse only works when 'mouse' contains 'n' or 'a'.

["x]zp or zp zP
["x]zP Like "p" and "P", except without adding trailing spaces when pasting a block. Thus the inserted text will not always be a rectangle. Especially useful in combination with v_zy.

You can use these commands to copy text from one place to another. Do this by first getting the text into a register with a yank, delete or change command, then inserting the register contents with a put command. You can also use these commands to move text from one file to another, because Vim preserves all registers when changing buffers (the CTRL-^ command is a quick way to toggle between two files).

linewise-register charwise-register
You can repeat the put commands with "." (except for :put) and undo them. If the command that was used to get the text into the register was linewise, Vim inserts the text below ("p") or above ("P") the line where the cursor is. Otherwise Vim inserts the text after ("p") or before ("P") the cursor. With the ":put" command, Vim always inserts the text in the next line. You can exchange two characters with the command sequence "xp". You can exchange two lines with the command sequence "ddp". You can exchange two words with the command sequence "deep" (start with the cursor in the blank space before the first word). You can use the `']` or `` `] `` command after the put command to move the cursor to the end of the inserted text, or use `'[` or `` `[ `` to move the cursor to the start.

put-Visual-mode

When using a put command like p or P in Visual mode, Vim will try to replace the selected text with the contents of the register. How this works depends on the type of selection and the text. With blockwise selection it also depends on the size of the block and whether the corners are on an existing character. (Implementation detail: it actually works by first putting the register after the selection and then deleting the selection.)

v_p

p in Visual mode puts text and sets the default register (unnamed, selection, or clipboard) to the previously-selected text. Useful if you want to put that text somewhere else. But you cannot repeat the same change.

v_P

P in Visual mode puts text without setting the default register. You can repeat the change, but the deleted text cannot be used. If you do need it you can use p with another register. E.g., yank the text to copy, Visually select the text to replace and use "0p . You can repeat this as many times as you like, and the unnamed register will be changed each time.

blockwise-put

When a register contains text from one line (characterwise), using a blockwise Visual selection, putting that register will paste that text

repeatedly in each of the selected lines, thus replacing the blockwise selected region by multiple copies of the register text. For example:

- yank the word "TEXT" into a register with yw
- select a visual block, marked with "v" in this text:

```
aaavvaaa
bbbvvbbb
cccvvccc
```

- press p, results in:

```
aaaTEXTaaa
bbbTEXTbbb
cccTEXTccc
```

blockwise-register

If you use a blockwise Visual mode command to get the text into the register, the block of text will be inserted before ("P") or after ("p") the cursor column in the current and next lines. Vim makes the whole block of text start in the same column. Thus the inserted text looks the same as when it was yanked or deleted. Vim may replace some <Tab> characters with spaces to make this happen. However, if the width of the block is not a multiple of a <Tab> width and the text after the inserted block contains <Tab>s, that text may be misaligned.

Use zP/|zp| to paste a blockwise yanked register without appending trailing spaces.

Note that after a charwise yank command, Vim leaves the cursor on the first yanked character that is closest to the start of the buffer. This means that "yl" doesn't move the cursor, but "yh" moves the cursor one character left.
Rationale: In Vi the "y" command followed by a backwards motion would sometimes not move the cursor to the first yanked character, because redisplaying was skipped. In Vim it always moves to the first character, as specified by Posix.
With a linewise yank command the cursor is put in the first line, but the column is unmodified, thus it may not be on the first yanked character.

Registers

registers {register} E354

There are ten types of registers:

1. The unnamed register ""
2. 10 numbered registers "0 to "9
3. The small delete register "-
4. 26 named registers "a to "z or "A to "Z
5. Three read-only registers ":, "., "%
6. Alternate buffer register "#
7. The expression register "=
8. The selection registers "* and "+
9. The black hole register "_
10. Last search pattern register "/

1. Unnamed register ""

quote_quote quotequote

Vim fills this register with text deleted with the "d", "c", "s", "x" commands or copied with the yank "y" command, regardless of whether or not a specific register was used (e.g. "xdd). This is like the unnamed register is pointing to the last used register. Thus when appending using an uppercase register name, the unnamed register contains the same text as the named register.
An exception is the '_' register: "_dd does not store the deleted text in any register.
Vim uses the contents of the unnamed register for any put command (p or P) which does not specify a register. Additionally you can access it with the name '""'. This means you have to type two double quotes. Writing to the "" register writes to register "0.

2. Numbered registers "0 to "9 quote_number

quote0 quote1 quote2 quote3 quote4 quote9

Vim fills these registers with text from yank and delete commands.
Numbered register 0 contains the text from the most recent yank command,
unless the command specified another register with ["x].

Numbered register 1 contains the text deleted by the most recent delete or
change command (even when the command specified another register), unless the
text is less than one line (the small delete register is used then). An
exception is made for the delete operator with these movement commands: %,
(,), `, /, ?, n, N, { and }.

Register "1 is always used then (this is Vi compatible). The "- register is
used as well if the delete is within a line. Note that these characters may
be mapped. E.g. % is mapped by the matchit plugin.

With each successive deletion or change, Vim shifts the previous contents
of register 1 into register 2, 2 into 3, and so forth, losing the previous
contents of register 9.

yankring

To also store yanks (not only deletions) in registers 1-9, try this:

```
-- Yank-ring: store yanked text in registers 1-9.
vim.api.nvim_create_autocmd('TextYankPost', {
  callback = function()
    if vim.v.event.operator == 'y' then
      for i = 9, 1, -1 do -- Shift all numbered registers.
        vim.fn.setreg(tostring(i), vim.fn.getreg(tostring(i - 1)))
      end
    end
  end,
})
```

3. Small delete register "- quote_- quote-

This register contains text from commands that delete less than one line,
except when the command specifies a register with ["x].

4. Named registers "a to "z or "A to "Z quote_alpha quotea

Vim fills these registers only when you say so. Specify them as lowercase
letters to replace their previous contents or as uppercase letters to append
to their previous contents. When the '>' flag is present in 'cpoptions' then
a line break is inserted before the appended text.

5. Read-only registers ":, ". and "%

These are '%', ':' and '.'. You can use them only with the "p", "P",
and ":put" commands and with CTRL-R.

quote_. quote. E29

- ". Contains the last inserted text (the same as what is inserted

  with the insert mode commands CTRL-A and CTRL-@). **Note:** this doesn't
  work with CTRL-R on the command-line. It works a bit differently,
  like inserting the text instead of putting it ('textwidth' and other
  options affect what is inserted).

quote_% quote%

- "% Contains the name of the current file.

quote_: quote: E30

- ": Contains the most recent executed command-line. Example: Use

  "@:" to repeat the previous command-line command. The command-line is
  only stored in this register when at least one character of it was
  typed. Thus it remains unchanged if the command was executed
  completely from a mapping.

6. Alternate file register "# quote_# quote# @#

Contains the alternate-file name for current window.
This register is writable with :let and setreg(). This changes which
buffer CTRL-^ enters. You can't yank or delete into this register.
A String is matched against existing buffer names, like :buffer:

```
let @# = 'buffer_name'
```

Also supports using buffer number and file-pattern.

Throws
  E86 when the buffer number does not exist
  E93 when more than one buffer matches
  E94 when none match

Clear the register with empty String:

```
let @# = ''
```

7. Expression register "=

quote_= quote= @=

This is not really a register that stores text, but is a way to use an expression in commands which use a register. The expression register is read-write.

- When typing the '=' after " or CTRL-R the cursor moves to the command-line,

  where you can enter any expression (see expression). All normal command-line editing commands are available, including a special history for expressions. When you end the command-line by typing <CR>, Vim computes the result of the expression. If you end it with <Esc>, Vim abandons the expression. If you do not enter an expression, Vim uses the previous expression (like with the "/" command).
- The expression must evaluate to a String. A Number is always automatically

  converted to a String. For the "p" and ":put" command, if the result is a Float it's converted into a String. If the result is a List each element is turned into a String and used as a line. A Dictionary is converted into a String. A Funcref results in an error message (use string() to convert).
- If the "= register is used for the "p" command, the String is split up at

  <NL> characters. If the String ends in a <NL>, it is regarded as a linewise register.

8. Selection registers "* and "+

Use these registers for storing and retrieving the selected text for the GUI. See quotestar and quoteplus. When the clipboard is not available or not working, the unnamed register is used instead. For Unix systems and Mac OS X, see primary-selection.

9. Black hole register "_

quote_

When writing to this register, nothing happens. This can be used to delete text without affecting the normal registers. When reading from this register, nothing is returned.

10. Last search pattern register "/

quote_/ quote/

Contains the most recent search-pattern. This is used for "n" and 'hlsearch'. This register is writable with :let and setreg(). You can change it to have 'hlsearch' highlight other matches without actually searching. You can't yank or delete into this register. The search direction is available in v:searchforward.
Note that the value is restored when returning from a function function-search-undo.

@/

You can write to a register with a :let command :let-@. Example:

```
:let @/ = "the"
```

If you use a put command without specifying a register, Vim uses the register that was last filled (this is also the contents of the unnamed register). If you are confused, use the :dis command to find out what Vim will put (this command displays all named and numbered registers; the unnamed register is labelled '"').

The next three commands always work on whole lines.

:[range]co[py] {address} :co :copy
Copy the lines given by [range] to below the line given by {address}.

:t

:t Synonym for copy.

:[range]m[ove] {address} :m :mo :move E134
Move the lines given by [range] to below the line given by {address}.

6. Formatting text

formatting

:[range]ce[nter] [width] :ce :center
Center lines in [range] between [width] columns (default 'textwidth' or 80 when 'textwidth' is 0).

:[range]ri[ght] [width] :ri :right
Right-align lines in [range] at [width] columns (default 'textwidth' or 80 when 'textwidth' is 0).

:le :left

:[range]le[ft] [indent]
Left-align lines in [range]. Sets the indent in the lines to [indent] (default 0).

gq

gq{motion} Format the lines that {motion} moves over.
Formatting is done with one of three methods:
1. If 'formatexpr' is not empty the expression is
evaluated. This can differ for each buffer.
2. If 'formatprg' is not empty an external program
is used.
3. Otherwise formatting is done internally.

In the third case the 'textwidth' option controls the
length of each formatted line (see below).
If the 'textwidth' option is 0, the formatted line
length is the screen width (with a maximum width of
79).
The 'formatoptions' option controls the type of
formatting fo-table.
The cursor is left on the first non-blank of the last
formatted line.

gqgq gqgq gqq
gqq Format the current line. With a count format that
many lines.

v_gq

{Visual}gq Format the highlighted text. (for {Visual} see
Visual-mode).

gw

gw{motion} Format the lines that {motion} moves over. Similar to
gq but puts the cursor back at the same position in
the text. However, 'formatprg' and 'formatexpr' are
not used.

gwgw gwgw gww
gww Format the current line as with "gw".

v_gw

{Visual}gw Format the highlighted text as with "gw". (for
{Visual} see Visual-mode).

Example: To format the current paragraph use: ggap

```
gqap
```

The "gq" command leaves the cursor in the line where the motion command takes the cursor. This allows you to repeat formatting repeated with ".". This works well with "gqj" (format current and next line) and "gq}" (format until end of paragraph). **Note:** When 'formatprg' is set, "gq" leaves the cursor on the first formatted line (as with using a filter command).

If you want to format the current paragraph and continue where you were, use:

```
gwap
```

If you always want to keep paragraphs formatted you may want to add the 'a' flag to 'formatoptions'. See auto-format.

If the 'autoindent' option is on, Vim uses the indent of the first line for the following lines.

Formatting does not change empty lines (but it does change lines with only white space!).

The 'joinspaces' option is used when lines are joined together.

You can set the 'formatexpr' option to an expression or the 'formatprg' option to the name of an external program for Vim to use for text formatting. The 'textwidth' and other options have no effect on formatting by an external program.

format-formatexpr

The 'formatexpr' option can be set to a Vim script function that performs reformatting of the buffer. This should usually happen in an ftplugin, since formatting is highly dependent on the type of file. It makes sense to use an autoload script, so the corresponding script is only loaded when actually needed and the script should be called <filetype>format.vim.

For example, the XML filetype plugin distributed with Vim in the $VIMRUNTIME/ftplugin directory, sets the 'formatexpr' option to:

```
setlocal formatexpr=xmlformat#Format()
```

That means, you will find the corresponding script, defining the xmlformat#Format() function, in the file $VIMRUNTIME/autoload/xmlformat.vim

Here is an example that removes trailing whitespace from the selected text.
Put it in your ftplugin:

```
vim.bo.formatexpr = function()
  -- only reformat on explicit gq command
  if vim.fn.mode() ~= 'n' then
    -- fall back to the internal reformatting
    return 1
  end
  local lines = vim.fn.getline(vim.v.lnum, vim.v.lnum + vim.v.count - 1)
  for i, val in ipairs(lines) do
    lines[i] = (val:gsub('%s+$', ''))
  end
  vim.fn.setline('.', lines)

  -- do not run internal formatter!
  return 0
end
```

Note: this function explicitly returns non-zero when called from insert mode (which basically means, text is inserted beyond the 'textwidth' limit). This causes Vim to fall back to reformat the text by using the internal formatter.

However, if the gq command is used to reformat the text, the function will receive the selected lines, trim trailing whitespace from those lines and put them back in place. If you are going to split single lines into multiple lines, be careful not to overwrite anything.

If you want to allow reformatting of text from insert or replace mode, one has to be very careful, because the function might be called recursively. For debugging it helps to set the 'debug' option.

right-justify

There is no command in Vim to right justify text. You can do it with an external command, like "par" (e.g.: :.,}!par to format until the end of the paragraph) or set 'formatprg' to "par".

format-comments

An overview of comment formatting is in section 30.6 of the user manual.

Vim can automatically insert and format comments in a special way. Vim recognizes a comment by a specific string at the start of the line (ignoring white space). Three types of comments can be used:

- A comment string that repeats at the start of each line. An example is the type of comment used in shell scripts, starting with "#".
- A comment string that occurs only in the first line, not in the following lines. An example is this list with dashes.
- Three-piece comments that have a start string, an end string, and optional lines in between. The strings for the start, middle and end are different. An example is the C style comment:

```
/*
 * this is a C comment
 */
```

The 'comments' option is a comma-separated list of parts. Each part defines a type of comment string. A part consists of:
        {flags}:{string}

{string} is the literal text that must appear.

{flags}:

n	Nested comment. Nesting with mixed parts is allowed. If 'comments' is "n:),n:>" a line starting with ">) >" is a comment.
b	Blank (<Space>, <Tab> or <EOL>) required after {string}.
f	Only the first line has the comment string. Do not repeat comment on the next line, but preserve indentation (e.g., a bullet-list).
s	Start of three-piece comment
m	Middle of a three-piece comment
e	End of a three-piece comment
l	Left align. Used together with 's' or 'e', the leftmost character of start or end will line up with the leftmost character from the middle. This is the default and can be omitted. See below for more details.
r	Right align. Same as above but rightmost instead of leftmost. See below for more details.
O	Don't consider this comment for the "O" command.
x	Allows three-piece comments to be ended by just typing the last character of the end-comment string as the first action on a new

line when the middle-comment string has been inserted automatically.
See below for more details.

{digits}
When together with 's' or 'e': add {digit} amount of offset to an automatically inserted middle or end comment leader. The offset begins from a left alignment. See below for more details.

-{digits}
Like {digits} but reduce the indent. This only works when there is some indent for the start or end part that can be removed.

When a string has none of the 'f', 's', 'm' or 'e' flags, Vim assumes the comment string repeats at the start of each line. The {flags} field may be empty.

Any blank space in the text before and after the {string} is part of the {string}, so do not include leading or trailing blanks unless the blanks are a required part of the comment string.

When one comment leader is part of another, specify the part after the whole. For example, to include both "-" and "->", use

```
:set comments=f:->,f:-
```

A three-piece comment must always be given as start,middle,end, with no other parts in between. An example of a three-piece comment is

```
sr:/*,mb:*,ex:*/
```

for C-comments. To avoid recognizing "*ptr" as a comment, the middle string includes the 'b' flag. For three-piece comments, Vim checks the text after the start and middle strings for the end string. If Vim finds the end string, the comment does not continue on the next line. Three-piece comments must have a middle string because otherwise Vim can't recognize the middle lines.

Notice the use of the "x" flag in the above three-piece comment definition. When you hit Return in a C-comment, Vim will insert the middle comment leader for the new line: " * ". To close this comment you just have to type "/" before typing anything else on the new line. This will replace the middle-comment leader with the end-comment leader and apply any specified alignment, leaving just " */". There is no need to hit Backspace first.

When there is a match with a middle part, but there also is a matching end part which is longer, the end part is used. This makes a C style comment work without requiring the middle part to end with a space.

Here is an example of alignment flags at work to make a comment stand out (kind of looks like a 1 too). Consider comment string:

```
:set comments=sr:/***,m:**,ex-2:******/
```

```
                                /***
                                  **<--right aligned from "r" flag
                                  **
offset 2 spaces for the "-2" flag-->**
                                ******/
```

In this case, the first comment was typed, then return was pressed 4 times, then "/" was pressed to end the comment.

Here are some finer points of three part comments. There are three times when alignment and offset flags are taken into consideration: opening a new line after a start-comment, opening a new line before an end-comment, and automatically ending a three-piece comment. The end alignment flag has a backwards perspective; the result is that the same alignment flag used with "s" and "e" will result in the same indent for the starting and ending pieces. Only one alignment per comment part is meant to be used, but an offset number will override the "r" and "l" flag.

Enabling 'cindent' will override the alignment flags in many cases. Reindenting using a different method like gg or = will not consult alignment flags either. The same behaviour can be defined in those other formatting options. One consideration is that 'cindent' has additional options for context based indenting of comments but cannot replicate many three piece indent alignments. However, 'indentexpr' has the ability to work better with three piece comments.

Other examples:

```
"b:*"   Includes lines starting with "*", but not if the "*" is
             followed by a non-blank.  This avoids a pointer dereference
             like "*str" to be recognized as a comment.
"n:>"   Includes a line starting with ">", ">>", ">>>", etc.
"fb:-"  Format a list that starts with "- ".
```

By default, "b:#" is included. This means that a line that starts with "#include" is not recognized as a comment line. But a line that starts with "# define" is recognized. This is a compromise.

fo-table

You can use the 'formatoptions' option to influence how Vim formats text.
'formatoptions' is a string that can contain any of the letters below. You
can separate the option letters with commas for readability.

letter	meaning when present in 'formatoptions'	
	fo-t	
t	Auto-wrap text using 'textwidth'	fo-c
c	Auto-wrap comments using 'textwidth', inserting the current comment leader automatically.	fo-r
r	Automatically insert the current comment leader after hitting <Enter> in Insert mode.	fo-o
o	Automatically insert the current comment leader after hitting 'o' or 'O' in Normal mode. In case comment is unwanted in a specific place use CTRL-U to quickly delete it. i_CTRL-U	fo-/
/	When 'o' is included: do not insert the comment leader for a // comment after a statement, only when // is at the start of the line.	fo-q
q	Allow formatting of comments with "gq". Note that formatting will not change blank lines or lines containing only the comment leader. A new paragraph starts after such a line, or when the comment leader changes.	fo-w
w	Trailing white space indicates a paragraph continues in the next line. A line that ends in a non-white character ends a paragraph.	fo-a
a	Automatic formatting of paragraphs. Every time text is inserted or deleted the paragraph will be reformatted. See auto-format. When the 'c' flag is present this only happens for recognized comments.	fo-n

n	When formatting text, recognize numbered lists. This actually uses the 'formatlistpat' option, thus any kind of list can be used. The indent of the text after the number is used for the next line. The default is to find a number, optionally followed by '.', ':', ')', ']' or '}'. Note that 'autoindent' must be set too. Doesn't work well together with "2". Example:
	`1. the first item`
	`  wraps`
	`2. the second item`	

fo-2

2 When formatting text, use the indent of the second line of a paragraph
for the rest of the paragraph, instead of the indent of the first
line. This supports paragraphs in which the first line has a
different indent than the rest. Note that 'autoindent' must be set
too. Example:

```
        first line of a paragraph
second line of the same paragraph
third line.
```

This also works inside comments, ignoring the comment leader.

fo-v

v Vi-compatible auto-wrapping in insert mode: Only break a line at a
blank that you have entered during the current insert command. (**Note:**
this is not 100% Vi compatible. Vi has some "unexpected features" or
bugs in this area. It uses the screen column instead of the line
column.)

fo-b

b Like 'v', but only auto-wrap if you enter a blank at or before
the wrap margin. If the line was longer than 'textwidth' when you
started the insert, or you do not enter a blank in the insert before
reaching 'textwidth', Vim does not perform auto-wrapping.

fo-l

l Long lines are not broken in insert mode: When a line was longer than
'textwidth' when the insert command started, Vim does not
automatically format it.

fo-m

m Also break at a multibyte character above 255. This is useful for
Asian text where every character is a word on its own. Note that
line breaks may also be added after punctuation characters such as
colons to match the CJK linebreaking rules.

fo-M

M When joining lines, don't insert a space before or after a multibyte
character. Overrules the 'B' flag.

fo-B

B When joining lines, don't insert a space between two multibyte
characters. Overruled by the 'M' flag.

fo-1

1 Don't break a line after a one-letter word. It's broken before it
instead (if possible).

fo-]

] Respect 'textwidth' rigorously. With this flag set, no line can be
longer than 'textwidth', unless line-break-prohibition rules make this
impossible. Mainly for CJK scripts and works only if 'encoding' is

"utf-8".

fo-j

j Where it makes sense, remove a comment leader when joining lines. For example, joining:

```
int i;   // the index
// in the list
```

Becomes:

```
int i;   // the index in the list
```

fo-p

p Don't break lines at single spaces that follow periods. This is intended to complement 'joinspaces' and cpo-J, for prose with sentences separated by two spaces. For example, with 'textwidth' set to 28:

```
Surely you're joking, Mr. Feynman!
```

Becomes:

```
Surely you're joking,
Mr. Feynman!
```

Instead of:

```
Surely you're joking, Mr.
Feynman!
```

With 't' and 'c' you can specify when Vim performs auto-wrapping:

value	action
""	no automatic formatting (you can use "gq" for manual formatting)
"t"	automatic formatting of text, but not comments
"c"	automatic formatting for comments, but not text (good for C code)
"tc"	automatic formatting for text and comments

Note that when 'textwidth' is 0, Vim does no automatic formatting anyway (but does insert comment leaders according to the 'comments' option). An exception is when the 'a' flag is present. auto-format

Note that 'textwidth' can be non-zero even if Vim never performs auto-wrapping; 'textwidth' is still useful for formatting with "gq".

If the 'comments' option includes "/*", "*" and/or "*/", then Vim has some built in stuff to treat these types of comments a bit more cleverly. Opening a new line before or after "/*" or "*/" (with 'r' or 'o' present in 'formatoptions') gives the correct start of the line automatically. The same happens with formatting and auto-wrapping. Opening a line after a line starting with "/*" or "*" and containing "*/", will cause no comment leader to be inserted, and the indent of the new line is taken from the line containing the start of the comment.
E.g.:

```
/*
 * Your typical comment.
 */
The indent on this line is the same as the start of the above
comment.
```

All of this should be really cool, especially in conjunction with the new :autocmd command to prepare different settings for different types of file.

Some examples:
for C code (only format comments):

```
:set fo=croq
```

for Mail/news (format all, don't start comment with "o" command):

```
:set fo=tcrq
```

Automatic formatting auto-format autoformat

When the 'a' flag is present in 'formatoptions' text is formatted automatically when inserting text or deleting text. This works nicely for editing text paragraphs. A few hints on how to use this:

- You need to properly define paragraphs. The simplest is paragraphs that are separated by a blank line. When there is no separating blank line, consider using the 'w' flag and adding a space at the end of each line in the paragraphs except the last one.
- You can set the 'formatoptions' based on the type of file filetype or specifically for one file with a modeline.
- Set 'formatoptions' to "aw2tq" to make text with indents like this:

```
    bla bla foobar bla
bla foobar bla foobar bla
```

bla bla foobar bla
bla foobar bla bla foobar

- Add the 'c' flag to only auto-format comments. Useful in source code.

- Set 'textwidth' to the desired width. If it is zero then 79 is used, or the

  width of the screen if this is smaller.

And a few warnings:

- When part of the text is not properly separated in paragraphs, making

  changes in this text will cause it to be formatted anyway. Consider doing

```
:set fo-=a
```

- When using the 'w' flag (trailing space means paragraph continues) and

  deleting the last line of a paragraph with dd, the paragraph will be
  joined with the next one.
- Changed text is saved for undo. Formatting is also a change. Thus each

  format action saves text for undo. This may consume quite a lot of memory.
- Formatting a long paragraph and/or with complicated indenting may be slow.

7. Sorting text sorting

Vim has a sorting function and a sorting command. The sorting function can be
found here: sort(), uniq().
Also see :uniq.

:sor :sort
:[range]sor[t][!] [b][f][i][l][n][o][r][u][x] [/{pattern}/]
Sort lines in [range]. When no range is given all
lines are sorted.

With [!] the order is reversed.

With [i] case is ignored.

:sort-l

With [l] sort uses the current collation locale.
Implementation details: strcoll() is used to compare
strings. See :language to check or set the collation
locale. Example:

```
:language collate en_US.UTF-8
:%sort l
```

v:collate can also used to check the current locale.
Sorting using the locale typically ignores case.
This does not work properly on Mac.

Options [n][f][x][o][b] are mutually exclusive.

With [n] sorting is done on the first decimal number
in the line (after or inside a {pattern} match).
One leading '-' is included in the number.

With [f] sorting is done on the Float in the line.
The value of Float is determined similar to passing
the text (after or inside a {pattern} match) to
str2float() function.

With [x] sorting is done on the first hexadecimal
number in the line (after or inside a {pattern}
match). A leading "0x" or "0X" is ignored.
One leading '-' is included in the number.

With [o] sorting is done on the first octal number in
the line (after or inside a {pattern} match).

With [b] sorting is done on the first binary number in
the line (after or inside a {pattern} match).

:sort-u :sort-uniq
With [u] (u stands for unique) only keep the first of
a sequence of identical lines (ignoring case when [i]
is used). Without this flag, a sequence of identical
lines will be kept in their original order.
Note that leading and trailing white space may cause
lines to be different.
When you just want to make things unique, use :uniq.

When /{pattern}/ is specified and there is no [r] flag
the text matched with {pattern} is skipped, so that
you sort on what comes after the match.
'ignorecase' applies to the pattern, but 'smartcase'
is not used.

Instead of the slash any non-letter can be used.
For example, to sort on the second comma-separated
field:

```
:sort /[^,]*,/
```

To sort on the text at virtual column 10 (thus
ignoring the difference between tabs and spaces):

```
:sort /.*\%10v/
```

To sort on the first number in the line, no matter
what is in front of it:

```
:sort /.\{-}\ze\d/
```

(Explanation: ".\{-}" matches any text, "\ze" sets the
end of the match and \d matches a digit.)
With [r] sorting is done on the matching {pattern}
instead of skipping past it as described above.
For example, to sort on only the first three letters
of each line:

```
:sort /\a\a\a/ r
```

If a {pattern} is used, any lines which don't have a
match for {pattern} are kept in their current order,
but separate from the lines which do match {pattern}.
If you sorted in reverse, they will be in reverse
order after the sorted lines, otherwise they will be
in their original order, right before the sorted
lines.

If {pattern} is empty (e.g. // is specified), the
last search pattern is used. This allows trying out
a pattern first.

Note that using :sort with :global doesn't sort the matching lines, it's
quite useless.

:sort does not use the current locale unless the l flag is used.
Vim does do a "stable" sort.

The sorting can be interrupted, but if you interrupt it too late in the
process you may end up with duplicated lines. This also depends on the system
library function used.

8. Deduplicating text

deduplicating unique

Vim has a deduplicating function and a deduplicating command. The
deduplicating function can be found here: unig().
Also see :sort-uniq.

:uni :unig

:[range]uni[q][!] [i][l][r][u] [/{pattern}/]
Remove duplicate lines that are adjacent to each other
in [range]. When no range is given, all lines are
processed.

With [i] case is ignored when comparing lines.

With [l] comparison uses the current collation locale.
See :sort-l for more details.

With [r] comparison is done on the text that matches
/{pattern}/ instead of the full line.

With [u] only keep lines that do not repeat (i.e., are
not immediately followed by the same line).

With [!] only keep lines that are immediately followed
by a duplicate.

If both [!] and [u] are given, [u] is ignored and [!]
takes effect.

When /{pattern}/ is specified and [r] is not used, the
text matched with {pattern} is skipped and comparison
is done on what comes after the match.
'ignorecase' applies to the pattern, but 'smartcase'
is not used.
Instead of the slash any non-letter can be used.

For example, to remove adjacent duplicate lines based
on the second comma-separated field:

```
:uniq /[^,]*,/
```

Or to keep only unique lines ignoring the first 5
characters:

```
:uniq u /.\{5}/
```

If {pattern} is empty (e.g. // is used), the last
search pattern is used.

Note that leading and trailing white space may cause
lines to be considered different.
To remove all duplicates regardless of position, use
:sort-u or external tools.

Repeating commands

Chapter 26 of the user manual introduces repeating usr_26.txt.

Single repeat

single-repeat

. dot-repeat

. Repeat last change, with count replaced with [count].
Also repeat a yank command, when the 'y' flag is
included in 'cpoptions'. Does not repeat a
command-line command.

@:

@: Repeat last command-line [count] times.

Simple changes (edits) can be repeated with the "." command. Without a count, the count of the last change is used. If you enter a count, it will replace the last one. v:count and v:count1 will be set.

If the last change included a specification of a numbered register, the register number will be incremented. See redo-register for an example how to use this.

Repeating an insert ins-repeat re-executes the whole session, including non-jump cursor-moves: "iab<Left>c<Esc>." produces "acb", not "c". Note that jumps (mouse, i_<PageUp>, …) split an insert session.

Repeating a Visual-mode command re-executes the captured keysequence, selection included. See visual-repeat.

To repeat non-changes, see motion-repeat. To repeat multiple actions, see cmdatom-macro and complex-repeat.

Semantic repeat

action-repeat cmdatom excalibur 🗡️

Every user action emits a CmdAtom event. This avoids the need for plugins to "announce" the repeatable unit, thus plugins like vim-repeat aren't needed.

CmdAtom.lhs is the high-level user input collected during an action, including getchar() input. This is signficant: it reflects the semantic intent. CmdAtom.keys reveals the low-level internal commands that were produced.

INPUT (lhs)	RESOLUTION (keys)	
dw	"dw"	Not translated.
zfa{	"zfa{"	Not an edit (changed=false).
x	"dl"	Translated builtin.
,d	"dl"	Mapping `:nnoremap ,d x`.
@q	"dl"	Macro `@q = "x"`.
<F6>	"dlw"	Composite `:nnoremap <F6> xw`: `type='mapping'`, subatoms "dl" and "w" concat to the composite keys.
,Dw	"dw"	Incomplete mapping: ended mid- operation (`:nnoremap ,D d`), captures its continuation ("w").
ds)	":call …<NL>)"	"Payload" mapping (e.g. vim-surround "ds'" reads getchar()).
]q	nil	Lua mapping.
V<C-E>d	""	Unreplayable: viewport-dependent Visual sequence; `lhs` is only a hint/label.

To repeat an atom, feed keys (mode "n"), or lhs (mode "m") if keys=nil.

```
local function replay(a)
  vim.api.nvim_feedkeys(a.keys or a.lhs, a.keys and 'n' or 'm', false)
end
```

motion-repeat

Example: This "," mapping repeats the last motion ("zj", "3w", "fx", "/pat<CR>", "]c", any mapping/command that sets moved=true).

```
local last ---@type vim.event.cmdatom.data?
vim.api.nvim_create_autocmd('CmdAtom', {
  callback = function(ev)
    -- Skip edits, and the "," mapping itself.
    local motion = ev.data.moved or ev.match == 'motion'
    if motion and not (ev.data.changed or ev.data.lhs == ',') then
      last = ev.data
    end
  end,
})
vim.keymap.set('n', ',', function()
  -- CmdAtom is deferred; schedule the replay, in case "," follows a motion.
  vim.schedule(function()
    if last then
      vim.api.nvim_feedkeys(last.keys or last.lhs, last.keys and 'n' or 'm', false)
    end
  end)
end)
```

edit-repeat

Example: This "." mapping repeats ANY edit (except undo/redo), including
operations provided by plugins, without the need for "announcement" via
vim-repeat or similar.

```
local last ---@type vim.event.cmdatom.data?
vim.api.nvim_create_autocmd('CmdAtom', {
  callback = function(ev)
    local is_redo_or_undo = ev.data.changed and (ev.data.undoseq or 0) <= (vim.b[ev.buf].maxseq or 0)
    vim.b[ev.buf].maxseq = math.max(vim.b[ev.buf].maxseq or 0, ev.data.undoseq or 0)
    if ev.data.changed and not is_redo_or_undo and ev.data.lhs ~= '.' then
      last = ev.data
    end
  end,
})
vim.keymap.set('n', '.', function()
  -- Multicursors: degrade to builtin "." (cascades).
  local mc = vim.api.nvim_create_namespace('nvim.multicursor')
  if #vim.api.nvim_buf_get_extmarks(0, mc, 0, -1, { limit = 1 }) > 0 then
    vim.api.nvim_feedkeys('.', 'n', false)
    return
  end
  -- CmdAtom is deferred; schedule the replay, in case "." follows an edit.
  vim.schedule(function()
    if last then
      vim.api.nvim_feedkeys(last.keys or last.lhs, last.keys and 'n' or 'm', false)
    end
  end)
end)
```

cmdatom-macro

Example: This "[count]<Space>" mapping saves the last [count] atoms as an
editable macro: "2<Space>" opens the cmdwin with the last [count] atoms
listed (you can delete/edit steps if you want), <Enter> saves it. "<Space>"
without a count replays the macro.

```
-- Track the last 20 atoms.
local atom_ring = {} ---@type vim.event.cmdatom.data[]
vim.api.nvim_create_autocmd('CmdAtom', {
  callback = function(ev)
    -- Skip this mapping itself, and cmdwin edits.
    if ev.data.lhs ~= ' ' and vim.fn.getcmdwintype() == '' then
      atom_ring[#atom_ring + 1] = ev.data
      if #atom_ring > 20 then
        table.remove(atom_ring, 1)
      end
    end
  end,
})
-- [count]<space> shows a cmdwin where the user can edit/save the last [count] atoms as a "macro".
-- <space> (no count) replays it.
vim.keymap.set('n', '<Space>', function()
  local count = vim.v.count
  -- CmdAtom is deferred; schedule it so pending events land in the ring first.
  vim.schedule(function()
    count = math.min(count, #atom_ring)
    if count == 0 then -- Replay the saved macro.
      for _, step in ipairs(vim.g.atom_macro or {}) do
        vim.api.nvim_feedkeys(vim.keycode(step.keys or step.lhs), step.keys and 'n' or 'm', false)
      end
      return
    end
    local parts = {}
    for i = #atom_ring - count + 1, #atom_ring do
      local a = atom_ring[i]
      local keys = a.keys or ('%s%s'):format(a.count or '', a.lhs)
      local field = a.keys and 'keys' or 'lhs'
      parts[#parts + 1] = ('{%s=%q},'):format(field, vim.fn.keytrans(keys))
    end
    local cmd = ('lua vim.g.atom_macro = { %s }'):format(table.concat(parts, ' '))
    -- Draft it on the cmdline; CTRL-F opens the cmdwin to edit it.
    vim.api.nvim_feedkeys((':%s%s'):format(cmd, vim.keycode('<C-f>')), 'n', false)
  end)
end)
```

restore-undo-cursor

By default, undo returns the cursor to its original position. If you don't
like that, you can use this snippet to get Vim's behavior instead:

```
vim.api.nvim_create_autocmd('CmdAtom', {
  callback = function(ev)
    -- Undo/redo: the buffer changed to an already-seen undo state.
    local undid = ev.data.changed and (ev.data.undoseq or 0) <= (vim.b[ev.buf].maxseq or 0)
    vim.b[ev.buf].maxseq = math.max(vim.b[ev.buf].maxseq or 0, ev.data.undoseq or 0)
    if undid then
      vim.cmd('normal! `[') -- Start of the changed text.
    end
  end,
})
```

Multiple repeat

multi-repeat

:g :global E148

:[range]g[lobal]/{pattern}/[cmd]
Execute the Ex command [cmd] (default ":p") on the
lines within [range] where {pattern} matches.

:[range]g[lobal]!/{pattern}/[cmd]
Execute the Ex command [cmd] (default ":p") on the
lines within [range] where {pattern} does NOT match.

:v :vglobal

:[range]v[global]/{pattern}/[cmd]
Same as :g!.

Example:

```
:g/^Obsolete/d _
```

Using the underscore after :d avoids clobbering registers or the clipboard.
This also makes it faster.

Instead of the '/' which surrounds the {pattern}, you can use any other
single byte character, but not an alphabetic character, '\', '"', '|' or '!'.
This is useful if you want to include a '/' in the search pattern or
replacement string.

For the definition of a pattern, see pattern.

NOTE [cmd] may contain a range; see collapse and edit-paragraph-join for
examples.

The global commands work by first scanning through the [range] lines and
marking each line where a match occurs (for a multi-line pattern, only the
start of the match matters).

In a second scan the [cmd] is executed for each marked line, as if the cursor was in that line. For ":v" and ":g!" the command is executed for each not marked line. If a line is deleted its mark disappears.
The default for [range] is the whole buffer (1,$). Use "CTRL-C" to interrupt the command. If an error message is given for a line, the command for that line is aborted and the global command continues with the next marked or unmarked line.

E147

When the command is used recursively, it only works on one line. Giving a range is then not allowed. This is useful to find all lines that match a pattern and do not match another pattern:

```
:g/found/v/notfound/{cmd}
```

This first finds all lines containing "found", but only executes {cmd} when there is no match for "notfound".

Any Ex command can be used, see ex-cmd-index. To execute a Normal mode command, you can use the :normal command:

```
:g/pat/normal {commands}
```

Make sure that {commands} ends with a whole command, otherwise Vim will wait for you to type the rest of the command for each match. The screen will not have been updated, so you don't know what you are doing. See :normal.

The undo/redo command will undo/redo the whole global command at once.
The previous context mark will only be set once (with "''" you go back to where the cursor was before the global command).

The global command sets both the last used search pattern and the last used substitute pattern (this is vi compatible). This makes it easy to globally replace a string:

```
:g/pat/s//PAT/g
```

This replaces all occurrences of "pat" with "PAT". The same can be done with:

```
:%s/pat/PAT/g
```

Which is two characters shorter!

Complex repeat

complex-repeat

q recording macro

q{0-9a-zA-Z"} Record typed characters into register {0-9a-zA-Z"} (uppercase to append). The 'q' command is disabled while executing a register, and it doesn't work inside a mapping and :normal.

Note: If the register being used for recording is also used for y and p the result is most likely not what is expected, because the put will paste the recorded macro and the yank will overwrite the recorded macro.

Note: The recording happens while you type, replaying the register happens as if the keys come from a mapping. This matters, for example, for undo, which only syncs when commands were typed.

q Stops recording.
Implementation note: The 'q' that stops recording is not stored in the register, unless it was the result of a mapping

@

@{0-9a-z".=*+} Execute the contents of register {0-9a-z".=*+} [count] times. Note that register '%' (name of the current file) and '#' (name of the alternate file) cannot be used.
The register is executed like a mapping, that means that the difference between 'wildchar' and 'wildcharm' applies, and undo might not be synced in the same way.
For "@=" you are prompted to enter an expression. The result of the expression is then executed.
See also @:.

@@ E748

@@ Repeat the previous @{0-9a-z":*} [count] times.

v_@-default

{Visual}@{0-9a-z".=*+}
{Visual}@@ In linewise Visual mode, execute the contents of the register for each selected line.
See visual-repeat, default-mappings.

:@

:[addr]@{0-9a-z".=*+} Execute the contents of register {0-9a-z".=*+} as an
                       Ex command. First set cursor at line [addr] (default
                       is current line). When the last line in the register
                       does not have a <CR> it will be added automatically
                       when the 'e' flag is present in 'cpoptions'.
                       For ":@=" the last used expression is used. The
                       result of evaluating the expression is executed as an
                       Ex command.
                       Mappings are not recognized in these commands.
                       When the line-continuation character (\) is present
                       at the beginning of a line in a linewise register,
                       then it is combined with the previous line. This is
                       useful for yanking and executing parts of a Vim
                       script.

                                                                    :@:

:[addr]@:   Repeat last command-line. First set cursor at line
                       [addr] (default is current line).

:[addr]@   :@@

:[addr]@@   Repeat the previous :@{register}. First set cursor at
                       line [addr] (default is current line).

Multiple cursors   mcursor multicursor

You can place extra cursors ("multicursor") in a buffer, to repeat operations at each cursor as-you-type. Unlike a macro (a stream of keys without any context), multicursor replays actions "semantically" (CmdAtom).

- Normal-mode commands, Mappings, and even Macros (@) replay at each cursor.
- Insert-mode edits appear at each cursor as you type.
- Motions (g=) and Visual sequences replay at each cursor: "viweex"
  re-executes the selection so the extents are per-cursor (each cursor's own word, block, …).
- Undo is atomic: u reverts the aggregate edits from all cursors at once.
- Registers are cursor-local. Each cursor reads/writes its own registers.
  The yanks are joined (linewise) when the multicursor session ends.
- Marks '< '> (gv), '['], '. '^ are cursor-local: "`." at each
  cursor goes to its own last change. The jumplist and changelist are global.
- Folds are ignored during replay. Edits replay within folds ("dd" deletes the
  line at cursor, not the entire folded contents). Motions step into folds instead of skipping over them. The primary cursor keeps the usual fold-behavior.
- Cursors are per-buffer, and cascade only while their buffer is current.
- Multicursors are presented as hl-MCursor highlights, unless your
  (terminal) UI supports the Kitty multiple-cursors protocol.
- CTRL-C interrupts the cascade.

Example: place a cursor on every matching pattern:

```
:g/pattern/normal! nQ
```

Example: place a cursor at each quickfix item, or a range:

```
:cdo normal! Q
:2,4cdo normal! Q
```

Example: place a cursor at every named mark:

```
zq]`
```

COMMANDS

                                                                    Q

Q   Toggles a multicursor at the current cursor position.
                       Disables 'follow' mode.

Use [count] to force (un)placement instead of toggling:
"1Q" adds, "2Q" removes.

zq

zq{motion} Places a cursor at {motion}, repeated from the current position until end of buffer (or limited to a Visual selection: v_zq). Think of it like %:g as an operator.

A search motion ("zq*", "zqn", "zqgn") places a cursor at every match, wrapping if 'wrapscan' is enabled. Sets the Visual area (gv) per-cursor, so "gv" selects the match at each cursor.

Examples:

- "zqw": every word.
- "zqgn": every match.
- "zq*gvc" changes each match of the word under the cursor.

Not supported:

- Text objects
- Lua, <Cmd> or ":" mappings (except :map-<expr>).

v_zq

{Visual}zq{motion} Like zq, but limited to the Visual range.

blockwise-visual acts on each line of the block, from the block's left column (or the end of a short line, like v_b_A).

Examples:

- "vipzqw": each word in the paragraph.
- "vipzqj" (or "zq+"): each line.
- "vipzq*": each match.
- "<C-V>3jzql": each line of the blockwise column.

v_Q

{Visual}Q Places a cursor on each line of the selection, at the cursor column. Same as "{Visual}zqj". v_zq
Enables 'follow' mode.

mcursor-mouse <C-LeftMouse>

<C-LeftMouse> Toggles a multicursor at the click position, without moving the primary cursor (or changing windows). No-op in Insert-mode.

q=

q= Toggles 'follow' mode: cursor-relative motions ("w", "fx", "*", …) are replayed per-cursor. Cursors arriving at the same position (e.g. "9j" on the last line) are merged (deduplicated).

Not replayed:

- Absolute motions (scrolls/jumps: "G", global marks, CTRL-O).
- Motions that fail/beep ("j" on the last line).

Use [count] to force the mode instead of toggling:
"1q=" on, "2q=" off.

mcursor-clear

CTRL-L Clears multicursors in the current buffer.
CTRL-L-default

Cursors can also be cleared by clearing the namespace:

```
local mc_ns = vim.api.nvim_create_namespace('nvim.multicursor')
vim.api.nvim_buf_clear_namespace(0, mc_ns, 0, -1)
```

gQ

gQ	Restores the previous multicursors, and the primary cursor position. No-op during a multicursor session.
	g_CTRL-A
g CTRL-A	During a multicursor session, inserts an ascending number ("counter") at each cursor, so a column of cursors becomes 1, 2, 3, …. Use [count] to choose the initial number.
	]C
]C	Jump to the [count]'th next cursor. Hint: "]CQ" removes cursors one at a time.
	[C
[C	Jump to the [count]'th previous cursor.
	Without a multicursor session, "]C" and "[C" jump to the previous session's cursors (gQ), and work as an exclusive motion: "d]C" deletes to the next one.

API mcursor-api

Plugins can place cursors by setting or deleting extmarks in the "nvim.multicursor" namespace, and inspect user actions via the CmdAtom event.

Adding an extmark (nvim_buf_set_extmark()) adds a cursor, unless one is already there. Query them with nvim_buf_get_extmarks(). Deleting an extmark deletes the cursor.

You can also use :marks to peek at cursors:

```
:marks nvim.multicursor
```

In rare cases a plugin may need to check v:cascading to change its per-cursor behavior. Such special handling is generally discouraged; multicursor is intended to work implicitly, without plugin modifications.

EXAMPLES mcursor-examples

Example: a "]C" mapping which places a cursor before jumping to the next, so you can "rotate" through the cursors.

```
local ns = vim.api.nvim_create_namespace('nvim.multicursor')
vim.keymap.set('n', ']C', function()
  local row, col = vim.pos.cursor(0):to_extmark()
  vim.api.nvim_buf_set_extmark(0, ns, row, col, {})
  vim.cmd('normal! ]C')
end)
```

Example: a "follow-once" mapping: "q-{motion}" replays {motion} at every cursor, then disables 'follow' mode again.

```
vim.keymap.set('n', 'q-', function()
  vim.cmd('normal! 1q=')
  vim.api.nvim_create_autocmd('CmdAtom', {
    callback = function(ev)
      if ev.data.lhs == 'q-' then
        return -- Skip the mapping itself.
      end
      vim.cmd('normal! 2q=')
      return true -- Delete the handler.
    end,
  })
end)
```

Example: place a cursor at every match, using the API:

```
local ns = vim.api.nvim_create_namespace('nvim.multicursor')
for _, m in ipairs(vim.fn.matchbufline('%', [[pattern]], 1, '$')) do
  vim.api.nvim_buf_set_extmark(0, ns, m.lnum - 1, m.byteidx, {})
end
```

LIMITATIONS mcursor-limitations

- During Insert-mode completion, live mirroring pauses if a preview could shift or overlap the primary's completion (same-line edits before the primary, or line insertions/deletions before it).
- During Insert-mode, edits by a <Cmd>/Lua/API are not replayed per-cursor; the primary text is mirrored as-is (similar to insert-completion).
- Cursor is removed if its Insert-entering operation fails. E.g. "ct;" on

a line without ";", "cgn" with no match.

- Explicit clipboard registers ("*yy, "+yy) are global, not per-cursor. On

  multicursor exit, you can set the clipboard to the joined yank:

```
:let @+=@"
```

- Implicit 'clipboard' ("unnamed"/"unnamedplus") updates the clipboard on

  multicursor exit.
- Same-line cursors may be lost/misplaced by an edit that shifts columns (e.g.

  inserting text).
- Time-travel undo (g-, g+, :earlier) clears all cursors.

- Reloading/unloading a buffer (:edit!, 'autoread') clears its cursors

  (extmarks limitation).
- Q, zq, gQ are not allowed while recording or executing a macro.

- zq at the start of a line guesses whether the start is a step: at the

  start of the buffer it always is ("ggVzqw" on an indented line), elsewhere
  "f"/"t" skip it ("0zqfa" on a line starting with "a").
- A mapping or function that does not re-read its input will not cascade

  correctly.

This file is about indenting C programs and other files.

1. Indenting C style programs

The basics for C style indenting are explained in section 30.2 of the user manual.

Vim has options for automatically indenting C style program files. Many programming languages including Java and C++ follow very closely the formatting conventions established with C. These options affect only the indent and do not perform other formatting. There are additional options that affect other kinds of formatting as well as indenting, see format-comments, fo-table, gq and formatting for the main ones.

There are in fact four main methods available for indentation, each one overrides the previous if it is enabled, or non-empty for 'indentexpr':

'autoindent'	uses the indent from the previous line.
'smartindent'	is like 'autoindent' but also recognizes some C syntax to increase/reduce the indent where appropriate.
'cindent'	Works more cleverly than the other two and is configurable to different indenting styles.
'indentexpr'	The most flexible of all: Evaluates an expression to compute the indent of a line. When non-empty this method overrides the other ones. See indent-expression.

The rest of this section describes the 'cindent' option.

Note that 'cindent' indenting does not work for every code scenario. Vim is not a C compiler: it does not recognize all syntax. One requirement is that toplevel functions have a "{" in the first column. Otherwise they are easily confused with declarations.

These five options control C program indenting:

'cindent'	Enables Vim to perform C program indenting automatically.
'cinkeys'	Specifies which keys trigger reindenting in insert mode.
'cinoptions'	Sets your preferred indent style.
'cinwords'	Defines keywords that start an extra indent in the next line.
'cinscopedecls'	Defines strings that are recognized as a C++ scope declaration.

If 'lisp' is not on and 'equalprg' is empty, the "=" operator indents using Vim's built-in algorithm rather than calling an external program.

See autocommand for how to set the 'cindent' option automatically for C code files and reset it for others.

cinkeys-format indentkeys-format

The 'cinkeys' option is a string that controls Vim's indenting in response to typing certain characters or commands in certain contexts. Note that this not only triggers C-indenting. When 'indentexpr' is not empty 'indentkeys' is used instead. The format of 'cinkeys' and 'indentkeys' is equal.

The default is "0{,0},0),0],:,0#,!^F,o,O,e" which specifies that indenting occurs as follows:

"0{"	if you type "{" as the first character in a line
"0}"	if you type "}" as the first character in a line
"0)"	if you type ")" as the first character in a line
"0]"	if you type "]" as the first character in a line
":"	if you type ":" after a label or case statement
"0#"	if you type "#" as the first character in a line
"!^F"	if you type CTRL-F (which is not inserted)
"o"	if you type a <CR> anywhere or use the "o" command (not in insert mode!)
"O"	if you use the "O" command (not in insert mode!)
"e"	if you type the second 'e' for an "else" at the start of a line

Characters that can precede each key: i_CTRL-F

!	When a "!" precedes the key, Vim will not insert the key but will instead reindent the current line. This allows you to define a command key for reindenting the current line. CTRL-F is the default key for this. Be careful if you define CTRL-I for this because CTRL-I is the ASCII code for <Tab>.
*	When a "*" precedes the key, Vim will reindent the line before inserting the key. If 'cinkeys' contains "*<Return>", Vim reindents the current line before opening a new line.
0	When a zero precedes the key (but appears after "!" or "*") Vim will reindent the line only if the key is the first character you type in the line. When used before "=" Vim will only reindent the line if there is only white space before the word.

When neither "!" nor "*" precedes the key, Vim reindents the line after you type the key. So ";" sets the indentation of a line which includes the ";".

Special key names:

>	Angle brackets mean spelled-out names of keys. For example: "<Up>", "<Ins>" (see key-notation).
^	Letters preceded by a caret (^) are control characters. For example: "^F" is CTRL-F.
o	Reindent a line when you use the "o" command or when Vim opens a new line below the current one (e.g., when you type <Enter> in insert mode).
O	Reindent a line when you use the "O" command.
e	Reindent a line that starts with "else" when you type the second 'e'.

:   Reindent a line when a ':' is typed which is after a label or case
        statement. Don't reindent for a ":" in "class::method" for C++. To
        Reindent for any ":", use "<:>".
=word   Reindent when typing the last character of "word". "word" may
        actually be part of another word. Thus "=end" would cause reindenting
        when typing the "d" in "endif" or "endwhile". But not when typing
        "bend". Also reindent when completion produces a word that starts
        with "word". "0=word" reindents when there is only white space before
        the word.
=~word Like =word, but ignore case.

If you really want to reindent when you type 'o', 'O', 'e', '0', '<', '>',
"*", ':' or '!', use "<o>", "<O>", "<e>", "<0>", "<<>", "<>>", "<*>", "<:>" or
"<!>", respectively, for those keys.

For an emacs-style indent mode where lines aren't indented every time you
press <Enter> but only if you press <Tab>, I suggest:

```
:set cinkeys=0{,0},:,0#,!<Tab>,!^F
```

You might also want to switch off 'autoindent' then.

Note: If you change the current line's indentation manually, Vim ignores the
cindent settings for that line. This prevents vim from reindenting after you
have changed the indent by typing <BS>, <Tab>, or <Space> in the indent or
used CTRL-T or CTRL-D.

cinoptions-values

The 'cinoptions' option sets how Vim performs indentation. The value after
the option character can be one of these (N is any number):
        N   indent N spaces
        -N   indent N spaces to the left
        Ns   N times 'shiftwidth' spaces
        -Ns   N times 'shiftwidth' spaces to the left

In the list below,
"N" represents a number of your choice (the number can be negative). When
there is an 's' after the number, Vim multiplies the number by 'shiftwidth':
"1s" is 'shiftwidth', "2s" is two times 'shiftwidth', etc. You can use a
decimal point, too: "-0.5s" is minus half a 'shiftwidth'.
The examples below assume a 'shiftwidth' of 4.

cino->

        >N   Amount added for "normal" indent. Used after a line that should
              increase the indent (lines starting with "if", an opening brace,
              etc.). (default 'shiftwidth').

```
  cino=               cino=>2             cino=>2s

if (cond)             if (cond)           if (cond)
{                     {                   {
    foo;                foo;                    foo;
}                     }                   }
```

cino-e

        eN   Add N to the prevailing indent inside a set of braces if the
              opening brace at the End of the line (more precise: is not the
              first character in a line). This is useful if you want a
              different indent when the "{" is at the start of the line from
              when "{" is at the end of the line. (default 0).

```
  cino=               cino=e2             cino=e-2

if (cond) {           if (cond) {         if (cond) {
    foo;                  foo;              foo;
}                     }                   }
else                  else                else
{                     {                   {
    bar;                  bar;                bar;
}                     }                   }
```

cino-n

        nN   Add N to the prevailing indent for a statement after an "if",
              "while", etc., if it is NOT inside a set of braces. This is
              useful if you want a different indent when there is no "{"
              before the statement from when there is a "{" before it.
              (default 0).

```
  cino=               cino=n2             cino=n-2

if (cond)             if (cond)           if (cond)
    foo;                  foo;              foo;
else                  else                else
{                     {                   {
    bar;                  bar;                bar;
}                     }                   }
```

cino-f

        fN   Place the first opening brace of a function or other block in

column N. This applies only for an opening brace that is not
inside other braces and is at the start of the line. What comes
after the brace is put relative to this brace. (default 0).

  cino=   cino=f.5s   cino=f1s

```
func()        func()              func()
{                       {                   {
    int foo;                int foo;            int foo;
```

cino-{

{N Place opening braces N characters from the prevailing indent.
This applies only for opening braces that are inside other
braces. (default 0).

  cino=   cino={.5s   cino={1s

```
if (cond)             if (cond)           if (cond)
{                       {                   {
    foo;                  foo;                foo;
```

cino-}

}N Place closing braces N characters from the matching opening
brace. (default 0).

  cino=   cino={2,}-0.5s   cino=}2

```
if (cond)             if (cond)           if (cond)
{                       {                 {
    foo;                  foo;                foo;
}                     }                     }
```

cino-^

^N Add N to the prevailing indent inside a set of braces if the
opening brace is in column 0. This can specify a different
indent for whole of a function (some may like to set it to a
negative number). (default 0).

  cino=   cino=^-2   cino=^-s

```
func()        func()              func()
{                     {                   {
    if (cond)           if (cond)         if (cond)
    {                   {                 {
        a = b;      a = b;              a = b;
    }                   }                 }
}                     }                   }
```

cino-L

LN Controls placement of jump labels. If N is negative, the label
will be placed at column 1. If N is non-negative, the indent of
the label will be the prevailing indent minus N. (default -1).

  cino=   cino=L2   cino=Ls

```
func()              func()              func()
{                   {                   {
    {                   {                   {
        stmt;               stmt;               stmt;
LABEL:                    LABEL:              LABEL:
    }                   }                   }
}                   }                   }
```

cino-:

:N Place case labels N characters from the indent of the switch().
(default 'shiftwidth').

  cino=   cino=:0

```
switch (x)            switch(x)
{                     {
    case 1:           case 1:
        a = b;    a = b;
    default:          default:
}                     }
```

cino-=

=N Place statements occurring after a case label N characters from
the indent of the label. (default 'shiftwidth').

  cino=   cino==10

```
case 11:                case 11:  a = a + 1;
    a = a + 1;            b = b + 1;
```

cino-l

lN	If N != 0 Vim will align with a case label instead of the
	statement after it in the same line.

	cino=	cino=l1

```
switch (a) {          switch (a) {
    case 1: {             case 1: {
               break;         break;
           }              }
```

cino-b

bN	If N != 0 Vim will align a final "break" with the case label,
	so that case..break looks like a sort of block. (default: 0).
	When using 1, consider adding "0=break" to 'cinkeys'.

	cino=	cino=b1

```
switch (x)            switch(x)
{                     {
    case 1:               case 1:
        a = b;        a = b;
        break;    break;

    default:              default:
        a = 0;        a = 0;
        break;    break;
}                     }
```

cino-g

gN	Place C++ scope declarations N characters from the indent of the
	block they are in. (default 'shiftwidth'). By default, a scope
	declaration is "public:", "protected:" or "private:". This can
	be adjusted with the 'cinscopedecls' option.

	cino=	cino=g0

```
{                     {
    public:           public:
        a = b;    a = b;
    private:          private:
}                     }
```

cino-h

hN	Place statements occurring after a C++ scope declaration N
	characters from the indent of the label. (default
	'shiftwidth').

	cino=	cino=h10

```
public:         public:   a = a + 1;
    a = a + 1;            b = b + 1;
```

cino-N

NN	Indent inside C++ namespace N characters extra compared to a
	normal block. (default 0).

	cino=	cino=N-s

```
namespace {               namespace {
    void function();      void function();
}                         }

namespace my              namespace my
{                         {
    void function();      void function();
}                         }
```

cino-E

EN	Indent inside C++ linkage specifications (extern "C" or
	extern "C++") N characters extra compared to a normal block.
	(default 0).

	cino=	cino=E-s

```
extern "C" {              extern "C" {
    void function();      void function();
}                         }

extern "C"                extern "C"
{                         {
    void function();      void function();
}                         }
```

cino-p

pN	Parameter declarations for K&R-style function declarations will
	be indented N characters from the margin. (default
	'shiftwidth').

```
              cino=               cino=p0             cino=p2s

func(a, b)            func(a, b)          func(a, b)
    int a;            int a;                      int a;
    char b;           char b;                     char b;
```

cino-t

tN Indent a function return type declaration N characters from the margin. (default 'shiftwidth').

```
              cino=               cino=t0             cino=t7

    int       int                       int
func()        func()             func()
```

cino-i

iN Indent C++ base class declarations and constructor initializations, if they start in a new line (otherwise they are aligned at the right side of the ':'). (default 'shiftwidth').

```
              cino=                   cino=i0

class MyClass :     class MyClass :
    public BaseClass      public BaseClass
{}                        {}
MyClass::MyClass() :      MyClass::MyClass() :
    BaseClass(3)          BaseClass(3)
{}                        {}
```

cino-+

+N Indent a continuation line (a line that spills onto the next) inside a function N additional characters. (default 'shiftwidth'). Outside of a function, when the previous line ended in a backslash, the 2 * N is used.

```
              cino=                   cino=+10

a = b + 9 *               a = b + 9 *
    c;                      c;
```

cino-c

cN Indent comment lines after the comment opener, when there is no other text with which to align, N characters from the comment opener. (default 3). See also format-comments.

```
              cino=                   cino=c5

/*                        /*
   text.                       text.
 */                        */
```

cino-C

CN When N is non-zero, indent comment lines by the amount specified with the c flag above even if there is other text behind the comment opener. (default 0).

```
              cino=c0                 cino=c0,C1

/********                  /********
  text.           text.
********/                 ********/
```

(Example uses ":set comments& comments-=s1:/* comments^=s0:/*")

cino-/

/N Indent comment lines N characters extra. (default 0).

```
              cino=                   cino=/4

a = b;            a = b;
/* comment */                 /* comment */
c = d;            c = d;
```

cino-(

(N When in unclosed parentheses, indent N characters from the line with the unclosed parenthesis. Add a 'shiftwidth' for every extra unclosed parentheses. When N is 0 or the unclosed parenthesis is the first non-white character in its line, line up with the next non-white character after the unclosed parenthesis. (default 'shiftwidth' * 2).

```
              cino=                   cino=(0
```

```
if (c1 && (c2 ||            if (c1 && (c2 ||
           c3))                        c3))
    foo;                        foo;
if (c1 &&                   if (c1 &&
        (c2 || c3))             (c2 || c3))
   {                           {
```

cino-u

uN Same as (N, but for one nesting level deeper. (default 'shiftwidth').

cino= cino=u2

```
if (c123456789      if (c123456789
        && (c22345                  && (c22345
            || c3))                   || c3))
```

cino-U

UN When N is non-zero, do not ignore the indenting specified by (or u in case that the unclosed parenthesis is the first non-white character in its line. (default 0).

cino= or cino=(s cino=(s,U1

```
c = c1 &&                   c = c1 &&
    (                           (
     c2 ||                          c2 ||
     c3                     c3
    ) && c4;                    ) && c4;
```

cino-w

wN When in unclosed parentheses and N is non-zero and either using "(0" or "u0", respectively, or using "U0" and the unclosed parenthesis is the first non-white character in its line, line up with the character immediately after the unclosed parenthesis rather than the first non-white character. (default 0).

cino=(0 cino=(0,w1

```
if (   c1                   if (   c1
       && (   c2                   && (   c2
              || c3))                     || c3))
    foo;                        foo;
```

cino-W

WN When in unclosed parentheses and N is non-zero and either using "(0" or "u0", respectively and the unclosed parenthesis is the last non-white character in its line and it is not the closing parenthesis, indent the following line N characters relative to the outer context (i.e. start of the line or the next unclosed parenthesis). (default: 0).

cino=(0 cino=(0,W4

```
a_long_line(                a_long_line(
            argument,                   argument,
            argument);  argument);
a_short_line(argument,    a_short_line(argument,
             argument);           argument);
```

cino-k

kN When in unclosed parentheses which follow "if", "for" or "while" and N is non-zero, overrides the behaviour defined by "(N": causes the indent to be N characters relative to the outer context (i.e. the line where "if", "for" or "while" is). Has no effect on deeper levels of nesting. Affects flags like "wN" only for the "if", "for" and "while" conditions. If 0, defaults to behaviour defined by the "(N" flag. (default: 0).

cino=(0 cino=(0,ks

```
if (condition1      if (condition1
    && condition2)                  && condition2)
    action();                   action();
function(argument1          function(argument1
         && argument2);              && argument2);
```

cino-m

mN When N is non-zero, line up a line starting with a closing parenthesis with the first character of the line with the matching opening parenthesis. (default 0).

cino=(s cino=(s,m1

```
c = c1 && (                 c = c1 && (
    c2 ||                       c2 ||
    c3                  c3
    ) && c4;                ) && c4;
if (                        if (
    c1 && c2                    c1 && c2
   )                        )
    foo;                        foo;
```

cino-M

MN When N is non-zero, line up a line starting with a closing parenthesis with the first character of the previous line. (default 0).

cino= cino=M1

```
if (cond1 &&                if (cond1 &&
       cond2                       cond2
   )                               )
```

java-cinoptions java-indenting cino-j

jN Indent Java anonymous classes correctly. Also works well for Javascript. The value 'N' is currently unused but must be non-zero (e.g. 'j1'). 'j1' will indent for example the following code snippet correctly:

```
object.add(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        do_something();
    }
});
```

javascript-cinoptions javascript-indenting cino-J

JN Indent JavaScript object declarations correctly by not confusing them with labels. The value 'N' is currently unused but must be non-zero (e.g. 'J1'). If you enable this you probably also want to set cino-j.

```
var bar = {
    foo: {
        that: this,
        some: ok,
    },
    "bar":{
        a : 2,
        b: "123abc",
        x: 4,
        "y": 5
    }
}
```

cino-)

)N Vim searches for unclosed parentheses at most N lines away. This limits the time needed to search for parentheses. (default 20 lines).

cino-star

N Vim searches for unclosed comments at most N lines away. This limits the time needed to search for the start of a comment. If your / */ comments stop indenting after N lines this is the value you will want to change. (default 70 lines).

cino-#

#N When N is non-zero recognize shell/Perl comments starting with '#', do not recognize preprocessor lines; allow right-shifting lines that start with "#". When N is zero (default): don't recognize '#' comments, do recognize preprocessor lines; right-shifting lines that start with "#" does not work.

cino-P

PN When N is non-zero recognize C pragmas, and indent them like any other code; does not concern other preprocessor directives. When N is zero (default): don't recognize C pragmas, treating them like every other preprocessor directive.

The defaults, spelled out in full, are:

```
cinoptions=>s,e0,n0,f0,{0,}0,^0,L-1,:s,=s,l0,b0,gs,hs,N0,E0,ps,ts,is,+s,
                c3,C0,/0,(2s,us,U0,w0,W0,k0,m0,j0,J0,)20,*70,#0,P0
```

Vim puts a line in column 1 if:

- It starts with '#' (preprocessor directives), if 'cinkeys' contains '#0'.

- It starts with a label (a keyword followed by ':', other than "case" and "default") and 'cinoptions' does not contain an 'L' entry with a positive value.
- Any combination of indentations causes the line to have less than 0 indentation.

2. Indenting by expression indent-expression

The basics for using flexible indenting are explained in section 30.3 of the user manual.

If you want to write your own indent file, it must set the 'indentexpr' option. Setting the 'indentkeys' option is often useful.
See the $VIMRUNTIME/indent/README.txt file for hints.
See the $VIMRUNTIME/indent directory for examples.

REMARKS ABOUT SPECIFIC INDENT FILES

CLOJURE ft-clojure-indent clojure-indent

Clojure indentation differs somewhat from traditional Lisps, due in part to the use of square and curly brackets, and otherwise by community convention. These conventions are not universally followed, so the Clojure indent script offers a few configuration options.

g:clojure_maxlines

Sets maximum scan distance of searchpairpos(). Larger values trade performance for correctness when dealing with very long forms. A value of 0 will scan without limits. The default is 300.

g:clojure_fuzzy_indent

g:clojure_fuzzy_indent_patterns

g:clojure_fuzzy_indent_blacklist

The 'lispwords' option is a list of comma-separated words that mark special forms whose subforms should be indented with two spaces.

For example:

```
(defn bad []
      "Incorrect indentation")

(defn good []
  "Correct indentation")
```

If you would like to specify 'lispwords' with a pattern instead, you can use the fuzzy indent feature:

```
" Default
let g:clojure_fuzzy_indent = 1
let g:clojure_fuzzy_indent_patterns = ['^with', '^def', '^let']
let g:clojure_fuzzy_indent_blacklist =
        \ ['-fn$', '\v^with-%(meta|out-str|loading-context)$']
```

g:clojure_fuzzy_indent_patterns and g:clojure_fuzzy_indent_blacklist are lists of patterns that will be matched against the unqualified symbol at the head of a list. This means that a pattern like "^foo" will match all these candidates: foobar, my.ns/foobar, and #'foobar.

Each candidate word is tested for special treatment in this order:

1. Return true if word is literally in 'lispwords'
2. Return false if word matches a pattern in g:clojure_fuzzy_indent_blacklist
3. Return true if word matches a pattern in g:clojure_fuzzy_indent_patterns
4. Return false and indent normally otherwise

g:clojure_special_indent_words

Some forms in Clojure are indented such that every subform is indented by only two spaces, regardless of 'lispwords'. If you have a custom construct that should be indented in this idiosyncratic fashion, you can add your symbols to the default list below.

```
" Default
let g:clojure_special_indent_words =
   \ 'deftype,defrecord,reify,proxy,extend-type,extend-protocol,letfn'
```

g:clojure_align_multiline_strings

Align subsequent lines in multi-line strings to the column after the opening
quote, instead of the same column.

For example:

```
(def default
  "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
  eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
  enim ad minim veniam, quis nostrud exercitation ullamco laboris
  nisi ut aliquip ex ea commodo consequat.")

(def aligned
  "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
   eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
   enim ad minim veniam, quis nostrud exercitation ullamco laboris
   nisi ut aliquip ex ea commodo consequat.")
```

g:clojure_align_subforms

By default, parenthesized compound forms that look like function calls and
whose head subform is on its own line have subsequent subforms indented by
two spaces relative to the opening paren:

```
(foo
  bar
  baz)
```

Setting this option to 1 changes this behaviour so that all subforms are
aligned to the same column, emulating the default behaviour of
clojure-mode.el:

```
(foo
 bar
 baz)
```

FORTRAN ft-fortran-indent

Block if, select case, select type, select rank, where, forall, type,
interface, associate, block, enum, critical, and change team constructs are
indented. The indenting of subroutines, functions, modules, and program
blocks is optional. Comments, labeled statements, and continuation lines are
indented in free source form, whereas they are not indented in fixed source
form because of the left margin requirements. Hence manual indent corrections
will be necessary for labeled statements and continuation lines when fixed
source form is being used. For further discussion of the method used for the
detection of source format see ft-fortran-syntax.

Do loops
All do loops are left unindented by default. Do loops can be unstructured in
Fortran with (possibly multiple) loops ending on a labeled executable
statement of almost arbitrary type. Correct indentation requires
compiler-quality parsing. Old code with do loops ending on labeled statements
of arbitrary type can be indented with elaborate programs such as Tidy.
Structured do/continue loops are also left unindented because continue
statements are used for purposes other than ending a do loop. Programs such
as Tidy can convert structured do/continue loops to the do/enddo form. Do
loops of the do/enddo variety can be indented. If you use only structured
loops of the do/enddo form, you should declare this by setting the
fortran_do_enddo variable in your vimrc as follows

```
let fortran_do_enddo=1
```

in which case do loops will be indented. If all your loops are of do/enddo
type only in, say, .f90 files, then you should set a buffer flag with an
autocommand such as

```
au! BufRead,BufNewFile *.f90 let b:fortran_do_enddo=1
```

to get do loops indented in .f90 files and left alone in Fortran files with
other extensions such as .for.

Program units
Indenting of program units (subroutines, functions, modules, and program
blocks) can be increased by setting the variable fortran_indent_more and can
be decreased by setting the variable fortran_indent_less. These variables
can be set for all fortran files in your vimrc as follows

```
let fortran_indent_less=1
```

A finer level of control can be achieved by setting the corresponding
buffer-local variable as follows

```
let b:fortran_indent_less=1
```

HTML ft-html-indent html-indent html-indenting

This is about variables you can set in your vimrc to customize HTML indenting.

You can set the indent for the first line after <script> and <style>
"blocktags" (default "zero"):

```
:let g:html_indent_script1 = "inc"
:let g:html_indent_style1 = "inc"
```

VALUE	MEANING
"zero"	zero indent
"auto"	auto indent (same indent as the blocktag)
"inc"	auto indent + one indent step

You can set the indent for attributes after an open <tag line:

```
:let g:html_indent_attribute = 1
```

VALUE	MEANING
1	auto indent, one indent step more than <tag
2	auto indent, two indent steps (default)
> 2	auto indent, more indent steps

Many tags increase the indent for what follows per default (see "Add Indent
Tags" in the script). You can add further tags with:

```
:let g:html_indent_inctags = "html,body,head,tbody"
```

You can also remove such tags with:

```
:let g:html_indent_autotags = "th,td,tr,tfoot,thead"
```

Default value is empty for both variables. **Note:** the initial "inctags" are
only defined once per Vim session.

User variables are only read when the script is sourced. To enable your
changes during a session, without reloading the HTML file, you can manually
do:

```
:call HtmlIndent_CheckUserSettings()
```

Detail:
  Calculation of indent inside "blocktags" with "alien" content:

```
BLOCKTAG   INDENT EXPR      WHEN APPLICABLE
      <script> : {customizable}     if first line of block
               : cindent(v:lnum)    if attributes empty or contain "java"
               : -1                 else (vbscript, tcl, ...)
      <style>  : {customizable}     if first line of block
               : GetCSSIndent()     else
      <!-- --> : -1
```

IDRIS2 ft-idris2-indent

Idris 2 indentation can be configured with several variables that control the
indentation level for different language constructs:

The "g:idris2_indent_if" variable controls the indentation of then and else
blocks after if statements. Defaults to 3.

The "g:idris2_indent_case" variable controls the indentation of patterns in
case expressions. Defaults to 5.

The "g:idris2_indent_let" variable controls the indentation after let
bindings. Defaults to 4.

The "g:idris2_indent_rewrite" variable controls the indentation after rewrite
expressions. Defaults to 8.

The "g:idris2_indent_where" variable controls the indentation of where
blocks. Defaults to 6.

The "g:idris2_indent_do" variable controls the indentation in do blocks.
Defaults to 3.

Example configuration:

```
let g:idris2_indent_if = 2
let g:idris2_indent_case = 4
let g:idris2_indent_let = 4
let g:idris2_indent_rewrite = 8
let g:idris2_indent_where = 6
let g:idris2_indent_do = 3
```

MATLAB ft-matlab-indent matlab-indent matlab-indenting

The setting Function indenting format in MATLAB Editor/Debugger Language
Preferences corresponds to:

```
:let g:MATLAB_function_indent = {0, 1 or 2 (default)}
```

Where 0 is for Classic, 1 for Indent nested functions and 2 for Indent all functions.

PHP ft-php-indent php-indent php-indenting

NOTE: PHP files will be indented correctly only if PHP syntax is active.

If you are editing a file in Unix 'fileformat' and '\r' characters are present before new lines, indentation won't proceed correctly ; you have to remove those useless characters first with a command like:

```
:%s /\r$//g
```

Or, you can simply :let the variable PHP_removeCRwhenUnix to 1 and the script will silently remove them when Vim loads a PHP file (at each BufRead).

OPTIONS:

PHP indenting can be altered in several ways by modifying the values of some global variables:

php-comment PHP_autoformatcomment

To not enable auto-formatting of comments by default (if you want to use your own 'formatoptions'):

```
:let g:PHP_autoformatcomment = 0
```

Else, 't' will be removed from the 'formatoptions' string and "qrowcb" will be added, see fo-table for more information.

PHP_outdentSLComments

To add extra indentation to single-line comments:

```
:let g:PHP_outdentSLComments = N
```

With N being the number of 'shiftwidth' to add.

Only single-line comments will be affected such as:

```
# Comment
// Comment
/* Comment */
```

PHP_default_indenting

To add extra indentation to every PHP lines with N being the number of 'shiftwidth' to add:

```
:let g:PHP_default_indenting = N
```

For example, with N = 1, this will give:

```
<?php
    if (!isset($History_lst_sel))
        if (!isset($History_lst_sel))
            if (!isset($History_lst_sel)) {
                $History_lst_sel=0;
            } else
                $foo="bar";

    $command_hist = TRUE;
?>
```

(Notice the extra indentation between the PHP container markers and the code)

PHP_outdentphpescape

To indent PHP escape tags as the surrounding non-PHP code (only affects the PHP escape tags):

```
:let g:PHP_outdentphpescape = 0
```

PHP_removeCRwhenUnix

To automatically remove '\r' characters when the 'fileformat' is set to Unix:

```
:let g:PHP_removeCRwhenUnix = 1
```

PHP_BracesAtCodeLevel

To indent braces at the same level than the code they contain:

```
:let g:PHP_BracesAtCodeLevel = 1
```

This will give the following result:

```
if ($foo)
    {
    foo();
    }
```

Instead of:

```
if ($foo)
{
    foo();
}
```

NOTE: Indenting will be a bit slower if this option is used because some optimizations won't be available.

PHP_vintage_case_default_indent

To indent 'case:' and 'default:' statements in switch() blocks:

```
:let g:PHP_vintage_case_default_indent = 1
```

In PHP braces are not required inside 'case/default' blocks therefore 'case:' and 'default:' are indented at the same level than the 'switch()' to avoid meaningless indentation. You can use the above option to return to the traditional way.

PHP_noArrowMatching

By default the indent script will indent multi-line chained calls by matching the position of the '->':

```
$user_name_very_long->name()
                    ->age()
                    ->info();
```

You can revert to the classic way of indenting by setting this option to 1:

```
:let g:PHP_noArrowMatching = 1
```

You will obtain the following result:

```
$user_name_very_long->name()
    ->age()
    ->info();
```

PHP_IndentFunctionCallParameters

Extra indentation levels to add to parameters in multi-line function calls.

```
let g:PHP_IndentFunctionCallParameters = 1
```

Function call arguments will indent 1 extra level. For two-space indentation:

```
function call_the_thing(
  $with_this,
  $and_that
) {
  $this->do_the_thing(
      $with_this,
      $and_that
  );
}
```

PHP_IndentFunctionDeclarationParameters

Extra indentation levels to add to arguments in multi-line function definitions.

```
let g:PHP_IndentFunctionDeclarationParameters = 1
```

Function arguments in declarations will indent 1 extra level. For two-space indentation:

```
function call_the_thing(
    $with_this,
    $and_that
) {
  $this->do_the_thing(
    $with_this,
    $and_that
  );
}
```

PYTHON ft-python-indent

The amount of indent can be set with the g:python_indent Dictionary, which
needs to be created before adding the items:

```
let g:python_indent = {}
```

The examples given are the defaults. Note that the dictionary values are set
to an expression, so that you can change the value of 'shiftwidth' later
without having to update these values.

Indent after an open paren:

```
let g:python_indent.open_paren = 'shiftwidth() * 2'
```

Indent after a nested paren:

```
let g:python_indent.nested_paren = 'shiftwidth()'
```

Indent for a continuation line:

```
let g:python_indent.continue = 'shiftwidth() * 2'
```

By default, the closing paren on a multiline construct lines up under the
first non-whitespace character of the previous line.
If you prefer that it's lined up under the first character of the line that
starts the multiline construct, reset this key:

```
let g:python_indent.closed_paren_align_last_line = v:false
```

The method uses searchpair() to look back for unclosed parentheses. This
can sometimes be slow, thus it timeouts after 150 msec. If you notice the
indenting isn't correct, you can set a larger timeout in msec:

```
let g:python_indent.searchpair_timeout = 500
```

If looking back for unclosed parenthesis is still too slow, especially during
a copy-paste operation, or if you don't need indenting inside multi-line
parentheses, you can completely disable this feature:

```
let g:python_indent.disable_parentheses_indenting = 1
```

For backward compatibility, these variables are also supported:

```
g:pyindent_open_paren
g:pyindent_nested_paren
g:pyindent_continue
g:pyindent_searchpair_timeout
g:pyindent_disable_parentheses_indenting
```

R ft-r-indent

Function arguments are aligned if they span for multiple lines. If you prefer
do not have the arguments of functions aligned, put in your vimrc:

```
let r_indent_align_args = 0
```

All lines beginning with a comment character, #, get the same indentation
level of the normal R code. Users of Emacs/ESS may be used to have lines
beginning with a single # indented in the 40th column, ## indented as R code,
and ### not indented. If you prefer that lines beginning with comment
characters are aligned as they are by Emacs/ESS, put in your vimrc:

```
let r_indent_ess_comments = 1
```

If you prefer that lines beginning with a single # are aligned at a column
different from the 40th one, you should set a new value to the variable
r_indent_comment_column, as in the example below:

```
let r_indent_comment_column = 30
```

Any code after a line that ends with "<-" is indented. Emacs/ESS does not
indent the code if it is a top-level function. If you prefer a behavior like
Emacs/ESS one in this regard, put in your vimrc:

```
let r_indent_ess_compatible = 1
```

Below is an example of indentation with and without this option enabled:

```
### r_indent_ess_compatible = 1
foo <-
    function(x)
{
    paste(x)
}
```

```
### r_indent_ess_compatible = 0
foo <-
    function(x)
    {
        paste(x)
    }
```

The code will be indented after lines that match the pattern
'\(&\|\|+\|-\|*\|/\|=\|\~\|%\|->\)\s*$'. If you want indentation after
lines that match a different pattern, you should set the appropriate value of
r_indent_op_pattern in your vimrc.

SHELL ft-sh-indent

The amount of indent applied under various circumstances in a shell file can
be configured by setting the following keys in the Dictionary
b:sh_indent_defaults to a specific amount or to a Funcref that references a
function that will return the amount desired:

b:sh_indent_options["default"] Default amount of indent.

b:sh_indent_options["continuation-line"]
                                Amount of indent to add to a continued line.

b:sh_indent_options["case-labels"]
                                Amount of indent to add for case labels.
                                (not actually implemented)

b:sh_indent_options["case-statements"]
                                Amount of indent to add for case statements.

b:sh_indent_options["case-breaks"]
                                Amount of indent to add (or more likely
                                remove) for case breaks.

VERILOG ft-verilog-indent

General block statements such as if, for, case, always, initial, function,
specify and begin, etc., are indented. The module block statements (first
level blocks) are not indented by default. you can turn on the indent with
setting a variable in the vimrc as follows:

```
let b:verilog_indent_modules = 1
```

then the module blocks will be indented. To stop this, remove the variable:

```
:unlet b:verilog_indent_modules
```

To set the variable only for Verilog file. The following statements can be
used:

```
au BufReadPost * if exists("b:current_syntax")
au BufReadPost *   if b:current_syntax == "verilog"
au BufReadPost *     let b:verilog_indent_modules = 1
au BufReadPost *   endif
au BufReadPost * endif
```

Furthermore, setting the variable b:verilog_indent_width to change the
indenting width (default is 'shiftwidth'):

```
let b:verilog_indent_width = 4
let b:verilog_indent_width = shiftwidth() * 2
```

In addition, you can turn the verbose mode for debug issue:

```
let b:verilog_indent_verbose = 1
```

Make sure to do ":set cmdheight=2" first to allow the display of the message.

VHDL ft-vhdl-indent

Alignment of generic/port mapping statements are performed by default. This
causes the following alignment example:

```
ENTITY sync IS
PORT (
       clk        : IN  STD_LOGIC;
       reset_n    : IN  STD_LOGIC;
       data_input : IN  STD_LOGIC;
       data_out   : OUT STD_LOGIC
     );
END ENTITY sync;
```

To turn this off, add

```
let g:vhdl_indent_genportmap = 0
```

to the vimrc file, which causes the previous alignment example to change:

```
ENTITY sync IS
PORT (
  clk        : IN  STD_LOGIC;
  reset_n    : IN  STD_LOGIC;
  data_input : IN  STD_LOGIC;
  data_out   : OUT STD_LOGIC
);
END ENTITY sync;
```

Alignment of right-hand side assignment "<=" statements are performed by
default. This causes the following alignment example:

```
sig_out <= (bus_a(1) AND
           (sig_b OR sig_c)) OR
           (bus_a(0) AND sig_d);
```

To turn this off, add

```
let g:vhdl_indent_rhsassign = 0
```

to the vimrc file, which causes the previous alignment example to change:

```
sig_out <= (bus_a(1) AND
  (sig_b OR sig_c)) OR
  (bus_a(0) AND sig_d);
```

Full-line comments (lines that begin with "--") are indented to be aligned
with the very previous line's comment, PROVIDED that a whitespace follows
after "--".

For example:

```
sig_a <= sig_b; -- start of a comment
                -- continuation of the comment
                -- more of the same comment
```

While in Insert mode, after typing "-- " (note the space " "), hitting CTRL-F
will align the current "-- " with the previous line's "--".

If the very previous line does not contain "--", THEN the full-line comment
will be aligned with the start of the next non-blank line that is NOT a
full-line comment.

Indenting the following code:

```
sig_c <= sig_d; -- comment 0
       -- comment 1
            -- comment 2
  --debug_code:
  --PROCESS(debug_in)
       --BEGIN
          --  FOR i IN 15 DOWNTO 0 LOOP
           --    debug_out(8*i+7 DOWNTO 8*i) <= debug_in(15-i);
          --  END LOOP;
   --END PROCESS debug_code;

    -- comment 3
sig_e <= sig_f; -- comment 4
         -- comment 5
```

results in:

```
sig_c <= sig_d; -- comment 0
                -- comment 1
                -- comment 2
--debug_code:
--PROCESS(debug_in)
--BEGIN
--  FOR i IN 15 DOWNTO 0 LOOP
--    debug_out(8*i+7 DOWNTO 8*i) <= debug_in(15-i);
--  END LOOP;
--END PROCESS debug_code;

-- comment 3
sig_e <= sig_f; -- comment 4
                -- comment 5
```

Notice that "--debug_code:" does not align with "-- comment 2"
because there is no whitespace that follows after "--" in "--debug_code:".

Given the dynamic nature of indenting comments, indenting should be done
TWICE. On the first pass, code will be indented. On the second pass,

full-line comments will be indented according to the correctly indented code.

VIM **ft-vim-indent**

g:vim_indent_cont

For indenting Vim scripts there is one variable that specifies the amount of
indent for a continuation line, a line that starts with a backslash:

```
:let g:vim_indent_cont = shiftwidth() * 3
```

Three times shiftwidth is the default value.

YAML **ft-yaml-indent**

By default, the yaml indent script does not try to detect multiline scalars.
If you want to enable this, set the following variable:

```
let g:yaml_indent_multiline_scalar = 1
```

Multi-byte support multi-byte

Chinese Japanese Korean

This is about editing text in languages which have many characters that can not be represented using one byte (one octet). Examples are Chinese, Japanese and Korean. Unicode is also covered here.

For an introduction to the most common features, see usr_45.txt in the user manual.
For changing the language of messages and menus see mlang.txt.

Getting started mbyte-first

This is a summary of the multibyte features in Nvim.

LOCALE

First of all, you must make sure your current locale is set correctly. If your system has been installed to use the language, it probably works right away. If not, you can often make it work by setting the $LANG environment variable in your shell:

```
setenv LANG ja_JP.EUC
```

Unfortunately, the name of the locale depends on your system. Japanese might also be called "ja_JP.EUCjp" or just "ja". To see what is currently used:

```
:language
```

To change the locale inside Vim use:

```
:language ja_JP.EUC
```

Vim will give an error message if this doesn't work. This is a good way to experiment and find the locale name you want to use. But it's always better to set the locale in the shell, so that it is used right from the start.

See mbyte-locale for details.

ENCODING

Nvim always uses UTF-8 internally. Thus 'encoding' is always set to "utf-8" and cannot be changed.

All the text that is used inside Vim will be in UTF-8. Not only the text in the buffers, but also in registers, variables, etc.

You can edit files in different encodings than UTF-8. Nvim will convert the file when you read it and convert it back when you write it.
See 'fileencoding', 'fileencodings' and ++enc.

DISPLAY AND FONTS

If you are working in a terminal (emulator) you must make sure it accepts UTF-8, the encoding which Vim is working with. Otherwise only ASCII can be displayed and edited correctly.

For the GUI you must select fonts that work with UTF-8. You can set 'guifont' and 'guifontwide'. 'guifont' is used for the single-width characters, 'guifontwide' for the double-width characters. Thus the 'guifontwide' font must be exactly twice as wide as 'guifont'. Example for UTF-8:

```
:set guifont=-misc-fixed-medium-r-normal-*-18-120-100-100-c-90-iso10646-1
:set guifontwide=-misc-fixed-medium-r-normal-*-18-120-100-100-c-180-iso10646-1
```

You can also set 'guifont' alone, the Nvim GUI will try to find a matching 'guifontwide' for you.

INPUT

There are several ways to enter multibyte characters:

- Your system IME can be used.

- Keymaps can be used. See mbyte-keymap.

The options 'iminsert', 'imsearch' and 'imcmdline' can be used to choose the different input methods or disable them temporarily.

Locale mbyte-locale

The easiest setup is when your whole system uses the locale you want to work in. But it's also possible to set the locale for one shell you are working in, or just use a certain locale inside Vim.

WHAT IS A LOCALE? locale

There are many languages in the world. And there are different cultures and environments at least as many as the number of languages. A linguistic

environment corresponding to an area is called "locale". This includes information about the used language, the charset, collating order for sorting, date format, currency format and so on. For Vim only the language and charset really matter.

You can only use a locale if your system has support for it. Some systems have only a few locales, especially in the USA. The language which you want to use may not be on your system. In that case you might be able to install it as an extra package. Check your system documentation for how to do that.

The location in which the locales are installed varies from system to system. For example, "/usr/share/locale" or "/usr/lib/locale". See your system's setlocale() man page.

Looking in these directories will show you the exact name of each locale. Mostly upper/lowercase matters, thus "ja_JP.EUC" and "ja_jp.euc" are different. Some systems have a locale.alias file, which allows translation from a short name like "nl" to the full name "nl_NL.ISO_8859-1".

Note that X-windows has its own locale stuff. And unfortunately uses locale names different from what is used elsewhere. This is confusing! For Vim it matters what the setlocale() function uses, which is generally NOT the X-windows stuff. You might have to do some experiments to find out what really works.

locale-name

The (simplified) format of locale name is:

```
        language
or      language_territory
or      language_territory.codeset
```

Territory means the country (or part of it), codeset means the charset. For example, the locale name "ja_JP.eucJP" means:

```
        ja      the language is Japanese
        JP      the country is Japan
        eucJP   the codeset is EUC-JP
```

But it also could be "ja", "ja_JP.EUC", "ja_JP.ujis", etc. And unfortunately, the locale name for a specific language, territory and codeset is not unified and depends on your system.

Examples of locale name:

charset	language	locale name
GB2312	Chinese (simplified)	zh_CN.EUC, zh_CN.GB2312
Big5	Chinese (traditional)	zh_TW.BIG5, zh_TW.Big5
CNS-11643	Chinese (traditional)	zh_TW
EUC-JP	Japanese	ja, ja_JP.EUC, ja_JP.ujis, ja_JP.eucJP
Shift_JIS	Japanese	ja_JP.SJIS, ja_JP.Shift_JIS
EUC-KR	Korean	ko, ko_KR.EUC

USING A LOCALE

To start using a locale for the whole system, see the documentation of your system. Mostly you need to set it in a configuration file in "/etc".

To use a locale in a shell, set the $LANG environment value. When you want to use Korean and the locale name is "ko", do this:

```
    sh:    export LANG=ko
    csh:   setenv LANG ko
```

You can put this in your ~/.profile or ~/.cshrc file to always use it.

To use a locale in Vim only, use the :language command:

```
:language ko
```

Put this in your init.vim file to use it always.

Or specify $LANG when starting Vim:

```
   sh:    LANG=ko vim {vim-arguments}
   csh:   env LANG=ko vim {vim-arguments}
```

You could make a small shell script for this.

Encoding

mbyte-encoding

UTF-8 is always used internally to encode characters. This applies to all the places where text is used, including buffers (files loaded into memory), registers and variables.

charset codeset

Charset is another name for encoding. There are subtle differences, but these don't matter when using Vim. "codeset" is another similar name.

Each character is encoded as one or more bytes. When all characters are encoded with one byte, we call this a single-byte encoding. The most often used one is called "latin1". This limits the number of characters to 256. Some of these are control characters, thus even fewer can be used for text.

When some characters use two or more bytes, we call this a multibyte encoding. This allows using much more than 256 characters, which is required for most East Asian languages.

Most multibyte encodings use one byte for the first 127 characters. These are equal to ASCII, which makes it easy to exchange plain-ASCII text, no

matter what language is used. Thus you might see the right text even when the
encoding was set wrong.

encoding-names

Vim can edit files in different character encodings. There are three major groups:

1   8bit   Single-byte encodings, 256 different characters. Mostly used
                in USA and Europe. Example: ISO-8859-1 (Latin1). All
                characters occupy one screen cell only.

2   2byte   Double-byte encodings, over 10000 different characters.
                Mostly used in Asian countries. Example: euc-kr (Korean)
                The number of screen cells is equal to the number of bytes
                (except for euc-jp when the first byte is 0x8e).

u   Unicode   Universal encoding, can replace all others. ISO 10646.
                Millions of different characters. Example: UTF-8. The
                relation between bytes and screen cells is complex.

Only UTF-8 is used by Vim internally. But files in other
encodings can be edited by using conversion, see 'fileencoding'.

Recognized 'fileencoding' values include: encoding-values

1   latin1   8-bit characters (ISO 8859-1, also used for cp1252)
1   iso-8859-n ISO_8859 variant (n = 2 to 15)
1   koi8-r   Russian
1   koi8-u   Ukrainian
1   macroman   MacRoman (Macintosh encoding)
1   8bit-{name} any 8-bit encoding (Vim specific name)
1   cp437   similar to iso-8859-1
1   cp737   similar to iso-8859-7
1   cp775   Baltic
1   cp850   similar to iso-8859-4
1   cp852   similar to iso-8859-1
1   cp855   similar to iso-8859-2
1   cp857   similar to iso-8859-5
1   cp860   similar to iso-8859-9
1   cp861   similar to iso-8859-1
1   cp862   similar to iso-8859-1
1   cp863   similar to iso-8859-8
1   cp865   similar to iso-8859-1
1   cp866   similar to iso-8859-5
1   cp869   similar to iso-8859-7
1   cp874   Thai
1   cp1250   Czech, Polish, etc.
1   cp1251   Cyrillic
1   cp1253   Greek
1   cp1254   Turkish
1   cp1255   Hebrew
1   cp1256   Arabic
1   cp1257   Baltic
1   cp1258   Vietnamese
1   cp{number} MS-Windows: any installed single-byte codepage
2   cp932   Japanese (Windows only)
2   euc-jp   Japanese
2   sjis   Japanese
2   cp949   Korean
2   euc-kr   Korean
2   cp936   simplified Chinese (Windows only)
2   euc-cn   simplified Chinese
2   cp950   traditional Chinese (alias for big5)
2   big5   traditional Chinese (alias for cp950)
2   euc-tw   traditional Chinese
2   2byte-{name} any double-byte encoding (Vim-specific name)
2   cp{number} MS-Windows: any installed double-byte codepage
u   utf-8   32 bit UTF-8 encoded Unicode (ISO/IEC 10646-1)
u   ucs-2   16 bit UCS-2 encoded Unicode (ISO/IEC 10646-1)
u   ucs-2le   like ucs-2, little endian
u   utf-16   ucs-2 extended with double-words for more characters
u   utf-16le   like utf-16, little endian
u   ucs-4   32 bit UCS-4 encoded Unicode (ISO/IEC 10646-1)
u   ucs-4le   like ucs-4, little endian

The {name} can be any encoding name that your system supports. It is passed
to iconv() to convert between UTF-8 and the encoding of the file.
For MS-Windows "cp{number}" means using codepage {number}.
Examples:

```
:set fileencoding=8bit-cp1252
:set fileencoding=2byte-cp932
```

The MS-Windows codepage 1252 is very similar to latin1. For practical reasons
the same encoding is used and it's called latin1.

Several aliases can be used, they are translated to one of the names above.
Incomplete list:

1   ansi   same as latin1 (obsolete, for backward compatibility)
2   japan   Japanese: "euc-jp"
2   korea   Korean: "euc-kr"
2   prc   simplified Chinese: "euc-cn"
2   chinese   same as "prc"

```
2   taiwan      traditional Chinese: "euc-tw"
u   utf8        same as utf-8
u   unicode     same as ucs-2
u   ucs2be      same as ucs-2 (big endian)
u   ucs-2be     same as ucs-2 (big endian)
u   ucs-4be     same as ucs-4 (big endian)
u   utf-32      same as ucs-4
u   utf-32le    same as ucs-4le
    default     the encoding of the current locale.
```

For the UCS codes the byte order matters. This is tricky, use UTF-8 whenever you can. The default is to use big-endian (most significant byte comes first):

```
name    bytes           char
            ucs-2             11 22         1122
            ucs-2le           22 11         1122
            ucs-4       11 22 33 44     11223344
            ucs-4le     44 33 22 11     11223344
```

On MS-Windows systems you often want to use "ucs-2le", because it uses little endian UCS-2.

There are a few encodings which are similar, but not exactly the same. Vim treats them as if they were different encodings, so that conversion will be done when needed. You might want to use the similar name to avoid conversion or when conversion is not possible:

```
        cp932, shift-jis, sjis
        cp936, euc-cn
```

CONVERSION charset-conversion

Vim will automatically convert from one to another encoding in several places:

- When reading a file and 'fileencoding' is different from "utf-8"
- When writing a file and 'fileencoding' is different from "utf-8"
- When displaying messages and the encoding used for LC_MESSAGES differs from "utf-8" (requires a gettext version that supports this).
- When reading a Vim script where :scriptencoding is different from "utf-8".

Most of these require iconv. Conversion for reading and writing files may also be specified with the 'charconvert' option.

Useful utilities for converting the charset:

```
    All:            iconv
        GNU iconv can convert most encodings.  Unicode is used as the
        intermediate encoding, which allows conversion from and to all other
        encodings.  See https://directory.fsf.org/wiki/Libiconv.
```

mbyte-conversion

When reading and writing files in an encoding different from "utf-8", conversion needs to be done. These conversions are supported:

- All conversions between Latin-1 (ISO-8859-1), UTF-8, UCS-2 and UCS-4 are handled internally.
- For MS-Windows, conversion from and to any codepage should work.
- Conversion specified with 'charconvert'
- Conversion with the iconv library, if it is available.

```
          Old versions of GNU iconv() may cause the conversion to fail (they
          request a very large buffer, more than Vim is willing to provide).
          Try getting another iconv() implementation.
```

Input with a keymap mbyte-keymap

When the keyboard doesn't produce the characters you want to enter in your text, you can use the 'keymap' option. This will translate one or more (English) characters to another (non-English) character. This only happens when typing text, not when typing Vim commands. This avoids having to switch between two keyboard settings.

The value of the 'keymap' option specifies a keymap file to use. The name of this file is one of these two:

```
        keymap/{keymap}_utf-8.vim
        keymap/{keymap}.vim
```

Here {keymap} is the value of the 'keymap' option.
The file name with "utf-8" included is tried first.

'runtimepath' is used to find these files. To see an overview of all
available keymap files, use this:

```
:echo globpath(&rtp, "keymap/*.vim")
```

In Insert and Command-line mode you can use CTRL-^ to toggle between using the
keyboard map or not. i_CTRL-^ c_CTRL-^
This flag is remembered for Insert mode with the 'iminsert' option. When
leaving and entering Insert mode the previous value is used. The same value
is also used for commands that take a single character argument, like f and
r.
For Command-line mode the flag is NOT remembered. You are expected to type an
Ex command first, which is ASCII.
For typing search patterns the 'imsearch' option is used. It can be set to
use the same value as for 'iminsert'.

lCursor

It is possible to give the GUI cursor another color when the language mappings
are being used. This is disabled by default, to avoid that the cursor becomes
invisible when you use a non-standard background color. Here is an example to
use a brightly colored cursor:

```
:highlight Cursor guifg=NONE guibg=Green
:highlight lCursor guifg=NONE guibg=Cyan
```

keymap-file-format :loadk :loadkeymap E105 E791
The keymap file looks something like this:

```
" Maintainer:   name <email@address>
" Last Changed: 2001 Jan 1

let b:keymap_name = "short"

loadkeymap
a       A
b       B       comment
```

The lines starting with a " are comments and will be ignored. Blank lines are
also ignored. The lines with the mappings may have a comment after the useful
text.

The "b:keymap_name" can be set to a short name, which will be shown in the
status line. The idea is that this takes less room than the value of
'keymap', which might be long to distinguish between different languages,
keyboards and encodings.

The actual mappings are in the lines below "loadkeymap". In the example "a"
is mapped to "A" and "b" to "B". Thus the first item is mapped to the second
item. This is done for each line, until the end of the file.
These items are exactly the same as what can be used in a :lmap command,
using "<buffer>" to make the mappings local to the buffer.
You can check the result with this command:

```
:lmap
```

The two items must be separated by white space. You cannot include white
space inside an item, use the special names "<Tab>" and "<Space>" instead.
The length of the two items together must not exceed 200 bytes.

It's possible to have more than one character in the first column. This works
like a dead key. Example:

```
'a      á
```

Since Vim doesn't know if the next character after a quote is really an "a",
it will wait for the next character. To be able to insert a single quote,
also add this line:

```
''      '
```

Since the mapping is defined with :lmap the resulting quote will not be
used for the start of another character defined in the 'keymap'.
It can be used in a standard :imap mapping.
The "accents" keymap uses this.

keymap-accents

The first column can also be in <> form:
```
        <C-c>           Ctrl-C
        <A-c>           Alt-c
        <A-C>           Alt-C
```
Note that the Alt mappings may not work, depending on your keyboard and
terminal.

Although it's possible to have more than one character in the second column,
this is unusual. But you can use various ways to specify the character:

```
A       a               literal character
A       <char-97>       decimal value
A       <char-0x61>     hexadecimal value
A       <char-0141>     octal value
x       <Space>         special key name
```

The characters are assumed to be encoded in UTF-8.
It's possible to use ":scriptencoding" when all characters are given
literally. That doesn't work when using the <char-> construct, because the
conversion is done on the keymap file, not on the resulting character.

The lines after "loadkeymap" are interpreted with 'cpoptions' set to "C".
This means that continuation lines are not used and a backslash has a special
meaning in the mappings. Examples:

```
" a comment line
\"      x       maps " to x
\\      y       maps \ to y
```

If you write a keymap file that will be useful for others, consider submitting
it to the Vim maintainer for inclusion in the distribution:
<maintainer@vim.org>

HEBREW KEYMAP keymap-hebrew

This file explains what characters are available in UTF-8 and CP1255
encodings, and what the keymaps are to get those characters:

glyph encoding	keymap
Char UTF-8 cp1255 hebrew hebrewp name	
0 אx5d0 0xe0 t	a alef
0 בx5d1 0xe1 c	b bet
0 גx5d2 0xe2 d	g gimel
0 דx5d3 0xe3 s	d dalet
0 הx5d4 0xe4 v	h he
0 וx5d5 0xe5 u	v vav
0 זx5d6 0xe6 z	z zayin
0 חx5d7 0xe7 j	j het
0 טx5d8 0xe8 y	T tet
0 יx5d9 0xe9 h	y yod
0 ךx5da 0xea l	K kaf sofit
0 כx5db 0xeb f	k kaf
0 לx5dc 0xec k	l lamed
0 מx5dd 0xed o	M mem sofit
0 מx5de 0xee n	m mem
0 ןx5df 0xef i	N nun sofit
0 נx5e0 0xf0 b	n nun
0 סx5e1 0xf1 x	s samech
0 עx5e2 0xf2 g	u ayin
0 ףx5e3 0xf3 ;	P pe sofit
0 פx5e4 0xf4 p	p pe
0 ץx5e5 0xf5 .	X tsadi sofit
0 צx5e6 0xf6 m	x tsadi
0 קx5e7 0xf7 e	q qof
0 רx5e8 0xf8 r	r resh
0 שx5e9 0xf9 a	w shin
0 תx5ea 0xfa ,	t tav

```
Vowel marks and special punctuation:
0    ְx5b0  0xc0     A:      A:   sheva
0    ֱx5b1  0xc1     HE      HE   hataf segol
0    ֲx5b2  0xc2     HA      HA   hataf patah
0    ֳx5b3  0xc3     HO      HO   hataf qamats
0    ִx5b4  0xc4     I       I    hiriq
0    ֵx5b5  0xc5     AY      AY   tsere
0    ֶx5b6  0xc6     E       E    segol
0    ַx5b7  0xc7     AA      AA   patah
0    ָx5b8  0xc8     AO      AO   qamats
0    ֹx5b9  0xc9     O       O    holam
0    ֻx5bb  0xcb     U       U    qubuts
0    ּx5bc  0xcc     D       D    dagesh
0    ֽx5bd  0xcd     ]T      ]T   meteg
0    ־x5be  0xce     ]Q      ]Q   maqaf
0    ֿx5bf  0xcf     ]R      ]R   rafe
0    ׀x5c0  0xd0     ]p      ]p   paseq
```

```
0    שׁx5c1  0xd1     SR      SR   shin-dot
0    שׂx5c2  0xd2     SL      SL   sin-dot
0    x5c3  0xd3     ]P      ]P   sof-pasuq
0    וּx5f0  0xd4     VV      VV   double-vav
0    וִx5f1  0xd5     VY      VY   vav-yod
0    יִx5f2  0xd6     YY      YY   yod-yod
```

The following are only available in UTF-8

Cantillation marks:

```
glyph
Char UTF-8 hebrew name
0    ֑x591   C:   etnahta
0    ֒x592   Cs   segol
0    ֓x593   CS   shalshelet
0    ֔x594   Cz   zaqef qatan
0    ֕x595   CZ   zaqef gadol
0    ֖x596   Ct   tipeha
0    ֗x597   Cr   revia
0    ֘x598   Cq   zarqa
0    ֙x599   Cp   pashta
0    ֚x59a   C!   yetiv
0    ֛x59b   Cv   tevir
0    ֜x59c   Cg   geresh
0    ֝x59d   C*   geresh qadim
0    ֞x59e   CG   gershayim
0    ֟x59f   CP   qarnei-parah
0    ֠x5aa   Cy   yerach-ben-yomo
0    ֫x5ab   Co   ole
0    ֬x5ac   Ci   iluy
0    ֭x5ad   Cd   dehi
0    ֮x5ae   Cn   zinor
0    ֯x5af   CC   masora circle
```

Combining forms:

```
0    ﬠxfb20  X`   Alternative ayin
0    ﬡxfb21  X'   Alternative alef
0    ﬢxfb22  X-d  Alternative dalet
0    ﬣxfb23  X-h  Alternative he
0    ﬤxfb24  X-k  Alternative kaf
0    ﬥxfb25  X-l  Alternative lamed
0    ﬦxfb26  X-m  Alternative mem-sofit
0    ﬧxfb27  X-r  Alternative resh
0    ﬨxfb28  X-t  Alternative tav
+    0xfb29  X-+  Alternative plus
0    שׁxfb2a  XW   shin+shin-dot
0    שׂxfb2b  Xw   shin+sin-dot
0    שּׁxfb2c  X..W  shin+shin-dot+dagesh
0    שּׂxfb2d  X..w  shin+sin-dot+dagesh
0    אַxfb2e  XA   alef+patah
0    אָxfb2f  XO   alef+qamats
0    אּxfb30  XI   alef+hiriq (mapiq)
0    בּxfb31  X.b  bet+dagesh
0    גּxfb32  X.g  gimel+dagesh
0    דּxfb33  X.d  dalet+dagesh
0    הּxfb34  X.h  he+dagesh
0    וּxfb35  Xu  vav+dagesh
0    זּxfb36  X.z  zayin+dagesh
0    טּxfb38  X.T  tet+dagesh
0    יּxfb39  X.y  yud+dagesh
0    ךּxfb3a  X.K  kaf sofit+dagesh
0    כּxfb3b  X.k  kaf+dagesh
0    לּxfb3c  X.l  lamed+dagesh
0    מּxfb3e  X.m  mem+dagesh
0    נּxfb40  X.n  nun+dagesh
0    סּxfb41  X.s  samech+dagesh
0    ףּxfb43  X.P  pe sofit+dagesh
0    פּxfb44  X.p  pe+dagesh
0    צּxfb46  X.x  tsadi+dagesh
0    קּxfb47  X.q  qof+dagesh
0    רּxfb48  X.r  resh+dagesh
0    שּxfb49  X.w  shin+dagesh
0    תּxfb4a  X.t  tav+dagesh
0    וֹxfb4b  Xo   vav+holam
0    בֿxfb4c  XRb  bet+rafe
0    כֿxfb4d  XRk  kaf+rafe
0    פֿxfb4e  XRp  pe+rafe
0    ﭏxfb4f  Xal  alef-lamed
```

Using UTF-8

mbyte-utf8 **UTF-8** **utf-8** **utf8**

Unicode unicode

The Unicode character set was designed to include all characters from other character sets. Therefore it is possible to write text in (almost) any language using Unicode. And it's mostly possible to mix these languages in one file, which is impossible with other encodings.

Unicode can be encoded in several ways. The most popular one is UTF-8, which uses one or more bytes for each character and is backwards compatible with ASCII. On MS-Windows UTF-16 is also used (previously UCS-2), which uses 16-bit words. Nvim supports all of these encodings, but always uses UTF-8 internally.

Nvim supports double-width characters; works best with 'guifontwide'. When using only 'guifont' the wide characters are drawn in the normal width and a space to fill the gap.

EMOJI

emoji

You can list emoji characters using this script:

```
:source $VIMRUNTIME/scripts/emoji_list.lua
```

bom-bytes

When reading a file a BOM (Byte Order Mark) can be used to recognize the Unicode encoding:

```
        EF BB BF     UTF-8
        FE FF        UTF-16 big endian
        FF FE        UTF-16 little endian
        00 00 FE FF  UTF-32 big endian
        FF FE 00 00  UTF-32 little endian
```

UTF-8 is the recommended encoding. Note that it's difficult to tell UTF-16 and UTF-32 apart. UTF-16 is often used on MS-Windows, UTF-32 is not widespread as file format.

mbyte-combining mbyte-composing

A composing or combining character is used to change the meaning of the character before it. The combining characters are drawn on top of the preceding character.

Nvim largely follows the definition of extended grapheme clusters in UAX#29 in the Unicode standard, with some modifications: An ascii char will always start a new cluster. In addition 'arabicshape' enables the combining of some arabic letters, when they are shaped to be displayed together in a single cell.

Too big combined characters cannot be displayed, but they can still be inspected using the g8 and ga commands described below.
When editing text a composing character is mostly considered part of the preceding character. For example "x" will delete a character and its following composing characters by default.
If the 'delcombine' option is on, then pressing 'x' will delete the combining characters, one at a time, then the base character. But when inserting, you type the first character and the following composing characters separately, after which they will be joined. The "r" command will not allow you to type a combining character, because it doesn't know one is coming. Use "R" instead.

Bytes which are not part of a valid UTF-8 byte sequence are handled like a single character and displayed as <xx>, where "xx" is the hex value of the byte.

Overlong sequences are not handled specially and displayed like a valid character. However, search patterns may not match on an overlong sequence. (an overlong sequence is where more bytes are used than required for the character.) An exception is NUL (zero) which is displayed as "<00>".

In the file and buffer the full range of Unicode characters can be used (31 bits). However, displaying only works for the characters present in the selected font.

Useful commands:

- "ga" shows the decimal, hexadecimal and octal value of the character under the cursor. If there are composing characters these are shown too. (If the message is truncated, use ":messages").
- "g8" shows the bytes used in a UTF-8 character, also the composing characters, as hex numbers.
- ":set fileencodings=" forces using UTF-8 for all files. The default is to automatically detect the encoding of a file.

STARTING VIM

You might want to select the font used for the menus. Unfortunately this doesn't always work. See the system specific remarks below, and 'langmenu'.

USING UTF-8 IN X-WINDOWS

utf-8-in-xwindows

You need to specify a font to be used. For double-wide characters another font is required, which is exactly twice as wide. There are two ways to do this:

1. Set 'guifont' and let Nvim find a matching 'guifontwide'
2. Set 'guifont' and 'guifontwide'

See the documentation for each option for details. Example:

```
:set guifont=-misc-fixed-medium-r-normal--15-140-75-75-c-90-iso10646-1
```

You might also want to set the font used for the menus. This only works for
Motif. Use the ":hi Menu font={fontname}" command for this. :highlight

TYPING UTF-8 utf-8-typing

If you are using X-Windows, you should find an input method that supports

UTF-8.

If your system does not provide support for typing UTF-8, you can use the
'keymap' feature. This allows writing a keymap file, which defines a UTF-8
character as a sequence of ASCII characters. See mbyte-keymap.

If everything else fails, you can type any character as four hex bytes:

```
CTRL-V u 1234
```

"1234" is interpreted as a hex number. You must type four characters, prepend
a zero if necessary.

COMMAND ARGUMENTS utf-8-char-arg

Commands like f, F, t and r take an argument of one character. For
UTF-8 this argument may include one or two composing characters. These need
to be produced together with the base character, Nvim doesn't wait for the next
character to be typed to find out if it is a composing character or not.
Using 'keymap' or :lmap is a nice way to type these characters.

The commands that search for a character in a line handle composing characters
as follows. When searching for a character without a composing character,
this will find matches in the text with or without composing characters. When
searching for a character with a composing character, this will only find
matches with that composing character. It was implemented this way, because
not everybody is able to type a composing character.

Overview of options mbyte-options

These options are relevant for editing multibyte files.

'fileencoding' Encoding of a file. When it's different from "utf-8"
                conversion is done when reading or writing the file.

'fileencodings' List of possible encodings of a file. When opening a file
                these will be tried and the first one that doesn't cause an
                error is used for 'fileencoding'.

'charconvert'   Expression used to convert files from one encoding to another.

'formatoptions' The 'm' flag can be included to have formatting break a line
                at a multibyte character of 256 or higher. Thus is useful for
                languages where a sequence of characters can be broken
                anywhere.

'keymap'   Specify the name of a keyboard mapping.

Contributions specifically for the multibyte features by:
        Chi-Deok Hwang <hwang@mizi.co.kr>
        SungHyun Nam <goweol@gmail.com>
        K.Nagano <nagano@atese.advantest.co.jp>
        Taro Muraoka <koron@tka.att.ne.jp>
        Yasuhiro Matsumoto <mattn@mail.goo.ne.jp>

Multi-language features multi-lang

This is about using messages and menus in various languages. For editing multibyte text see multibyte.

The basics are explained in the user manual: usr_45.txt.

1. Messages multilang-messages

Vim picks up the locale from the environment. In most cases this means Vim will use the language that you prefer, unless it's not available.

To see a list of supported locale names on your system, look in one of these directories (for Unix):
/usr/lib/locale
/usr/share/locale
Unfortunately, upper/lowercase differences matter. Also watch out for the use of "-" and "_".

:lan :lang :language E197

:lan[guage]
:lan[guage] mes[sages]
:lan[guage] cty[pe]
:lan[guage] tim[e]
:lan[guage] col[late]
Print the current language (aka locale).
With the "messages" argument the language used for messages is printed. Technical: LC_MESSAGES.
With the "ctype" argument the language used for character encoding is printed. Technical: LC_CTYPE.
With the "time" argument the language used for strftime() is printed. Technical: LC_TIME.
With the "collate" argument the language used for collation order is printed. Technical: LC_COLLATE.
Without argument all parts of the locale are printed (this is system dependent).
The current language can also be obtained with the v:lang, v:ctype, v:collate and v:lc_time variables.

:lan[guage] {name}
:lan[guage] mes[sages] {name}
:lan[guage] cty[pe] {name}
:lan[guage] tim[e] {name}
:lan[guage] col[late] {name}
Set the current language (aka locale) to {name}. The locale {name} must be a valid locale on your system. Some systems accept aliases like "en" or "en_US", but some only accept the full specification like "en_US.ISO_8859-1". On Unix systems you can use this command to see what locales are supported:

```
:!locale -a
```

With the "messages" argument the language used for messages is set. This can be different when you want, for example, English messages while editing Japanese text. This sets $LC_MESSAGES.
With the "ctype" argument the language used for character encoding is set. This affects the libraries that Vim was linked with. It's unusual to set this to a different value from 'encoding' or "C". This sets $LC_CTYPE.
With the "time" argument the language used for time and date messages is set. This affects strftime(). This sets $LC_TIME.
With the "collate" argument the language used for the collation order is set. This affects sorting of characters. This sets $LC_COLLATE.
Without an argument all are set, and additionally $LANG is set.
The LC_NUMERIC value will always be set to "C" so that floating point numbers use '.' as the decimal point. This will make a difference for items that depend on the language (some messages, time and date format).
Not fully supported on all systems.
If this fails there will be an error message. If it succeeds there is no message. Example:

```
:language
Current language: C
:language de_DE.ISO_8859-1
:language mes
Current messages language: de_DE.ISO_8859-1
:lang mes en
```

Message files (vim.mo) have to be placed in "$VIMRUNTIME/lang/xx/LC_MESSAGES", where "xx" is the abbreviation of the language (mostly two letters). If you write your own translations you need to generate the .po file and convert it to a .mo file.

To overrule the automatic choice of the language, set the $LANG variable to
the language of your choice. use "en" to disable translations.

```
:let $LANG = 'ja'
```

(text for Windows by Muraoka Taro)

2. Menus multilang-menus

See 45.2 for the basics, esp. using 'langmenu'.

Note that if changes have been made to the menus after the translation was
done, some of the menus may be shown in English. Please try contacting the
maintainer of the translation and ask him to update it. You can find the
name and e-mail address of the translator in
"$VIMRUNTIME/lang/menu_<lang>.vim".

To set the font to use for the menus, use the :highlight command. Example:

```
:highlight Menu font=k12,r12
```

ALIAS LOCALE NAMES

Unfortunately, the locale names are different on various systems, even though
they are for the same language and encoding. If you do not get the menu
translations you expected, check the output of this command:

```
echo v:lang
```

Now check the "$VIMRUNTIME/lang" directory for menu translation files that use
a similar language. A difference in a "-" being a "_" already causes a file
not to be found! Another common difference to watch out for is "iso8859-1"
versus "iso_8859-1". Fortunately Vim makes all names lowercase, thus you
don't have to worry about case differences. Spaces are changed to
underscores, to avoid having to escape them.

If you find a menu translation file for your language with a different name,
create a file in your own runtime directory to load that one. The name of
that file could be:

```
~/.config/nvim/lang/menu_<v:lang>.vim
```

Check the 'runtimepath' option for directories which are searched. In that
file put a command to load the menu file with the other name:

```
runtime lang/menu_<other_lang>.vim
```

TRANSLATING MENUS

If you want to do your own translations, you can use the :menutrans command,
explained below. It is recommended to put the translations for one language
in a Vim script. For a language that has no translation yet, please consider
becoming the maintainer and make your translations available to all Vim users.
Send an e-mail to the Vim maintainer <maintainer@vim.org>.

:menut :menutrans :menutranslate

:menut[ranslate] clear
Clear all menu translations.

:menut[ranslate] {english} {mylang}
Translate menu name {english} to {mylang}. All
special characters like "&" and "<Tab>" need to be
included. Spaces and dots need to be escaped with a
backslash, just like in other :menu commands.
Case in {english} is ignored.

See the $VIMRUNTIME/lang directory for examples.

To try out your translations you first have to remove all menus. This is how
you can do it without restarting Vim:

```
:source $VIMRUNTIME/delmenu.vim
:source <your-new-menu-file>
:source $VIMRUNTIME/menu.vim
```

Each part of a menu path is translated separately. The result is that when
"Help" is translated to "Hilfe" and "Overview" to "Überblick" then
"Help.Overview" will be translated to "Hilfe.Überblick".

3. Scripts multilang-scripts

In Vim scripts you can use the v:lang variable to get the current language
(locale). The default value is "C" or comes from the $LANG environment
variable.

The following example shows how this variable is used in a simple way, to make
a message adapt to language preferences of the user,

```
:if v:lang =~ "de_DE"
:  echo "Guten Morgen"
:else
:  echo "Good morning"
:endif
```

updated by Nadim Shaikli

Right to Left display mode for Vim

These functions were originally created by Avner Lottem:
E-mail: alottem@iil.intel.com
Phone: +972-4-8307322

Introduction

Some languages such as Arabic, Farsi, Hebrew (among others) require the ability to display their text from right-to-left. Files in those languages are stored conventionally and the right-to-left requirement is only a function of the display engine (per the Unicode specification). In right-to-left oriented files the characters appear on the screen from right to left.

Bidirectionality (or bidi for short) is what Unicode offers as a full solution to these languages. Bidi offers the user the ability to view both right-to-left as well as left-to-right text properly at the same time within the same window. Vim currently, due to simplicity, does not offer bidi and is merely opting to present a functional means to display/enter/use right-to-left languages. An older hybrid solution in which direction is encoded for every character (or group of characters) are not supported either as this kind of support is out of the scope of a simple addition to an existing editor (and it's not sanctioned by Unicode either).

As many people working on the code do not use the right-to-left mode, this feature may not work in some situations. If you can describe what is wrong and how it would work when fixed, please create an issue on github, see bug-report.

Highlights

- Editing left-to-right files as in the original Vim, no change.
- Viewing and editing files in right-to-left windows. File orientation is per window, so it is possible to view the same file in right-to-left and left-to-right modes, simultaneously. (Useful for editing mixed files in which both right-to-left and left-to-right text exist).
- Compatibility to the original Vim. Almost all features work in right-to-left mode (see Bugs below).
- Backing from reverse insert mode to the correct place in the file (if possible).
- No special terminal with right-to-left capabilities is required. The right-to-left changes are completely hardware independent.
- Many languages use and require right-to-left support. These languages can quite easily be supported given the inclusion of their required keyboard mappings and some possible minor code change. Some of the current supported languages include - l10n-arabic.txt and l10n-hebrew.txt.

Of Interest...

- Invocations
  + 'rightleft' ('rl') sets window orientation to right-to-left.
  + 'delcombine' ('deco'), boolean, if editing UTF-8 encoded languages, allows one to remove a composing character which gets superimposed on those that preceded them (some languages require this).
  + 'rightleftcmd' ('rlc') sets the command-line within certain modes (such as search) to be utilized in right-to-left orientation as well.
- Typing backwards ins-reverse

  In lieu of using the full-fledged 'rightleft' option, one can opt for reverse insertion. When the 'revins' (reverse insert) option is set, inserting happens backwards. This can be used to type right-to-left text. When inserting characters the cursor is not moved and the text moves rightwards. A <BS> deletes the character under the cursor. CTRL-W and CTRL-U also work in the opposite direction. <BS>, CTRL-W and CTRL-U do not stop at the start of insert or end of line, no matter how the 'backspace' option is set.

  There is no reverse replace mode (yet).

  If the 'showmode' option is set, "-- REVERSE INSERT --" will be shown in the status line when reverse Insert mode is active.
- Pasting when in a rightleft window

  When cutting text with the mouse and pasting it in a rightleft window the text will be reversed, because the characters come from the cut buffer from the left to the right, while inserted in the file from the right to the left. In order to avoid it, toggle 'revins' before pasting.

Bugs

- Does not handle CTRL-A and CTRL-X commands (add and subtract) correctly when in rightleft window.
- Does not support reverse insert and rightleft modes on the command-line. However, functionality of the editor is not reduced, because it is possible to enter mappings, abbreviations and searches typed from the left to the right on the command-line.

- Somewhat slower in right-to-left mode, because right-to-left motion is emulated inside Vim, not by the controlling terminal.
- When both 'rightleft' and 'revins' are on: 'textwidth' does not work. Lines do not wrap at all; you just get a single, long line.
- There is no full bidirectionality (bidi) support.

Arabic Language support (options & mappings) for Vim

These functions have been created by Nadim Shaikli <nadim-at-arabeyes.org>

It is best to view this file with these settings within VIM's GUI:

```
:set encoding=utf-8
:set arabicshape
```

Introduction

Arabic is a rather demanding language in which a number of special features are required. Characters are right-to-left oriented and ought to appear as such on the screen (i.e. from right to left). Arabic also requires shaping of its characters, meaning the same character has a different visual form based on its relative location within a word (initial, medial, final or stand-alone). Arabic also requires two different forms of combining and the ability, in certain instances, to either superimpose up to two characters on top of another (composing) or the actual substitution of two characters into one (combining). Lastly, to display Arabic properly one will require not only ISO-8859-6 (U+0600-U+06FF) fonts, but will also require Presentation Form-B (U+FE70-U+FEFF) fonts both of which are subsets within a so-called ISO-10646-1 font.

The commands, prompts and help files are not in Arabic, therefore the user interface remains the standard Vi interface.

Highlights

- Editing left-to-right files as in the original Vim hasn't changed.
- Viewing and editing files in right-to-left windows. File orientation is per window, so it is possible to view the same file in right-to-left and left-to-right modes, simultaneously.
- No special terminal with right-to-left capabilities is required. The right-to-left changes are completely hardware independent. Only Arabic fonts are necessary.
- Compatible with the original Vim. Almost all features work in right-to-left mode (there are liable to be bugs).
- Changing keyboard mapping and reverse insert modes using a single command.
- Toggling complete Arabic support via a single command.
- While in Arabic mode, numbers are entered from left to right. Upon entering a none number character, that character will be inserted just into the left of the last number.
- Arabic keymapping on the command line in reverse insert mode.
- Proper Bidirectional functionality is possible given Vim is started within a Bidi capable terminal emulator.

Arabic Fonts

arabicfonts

Vim requires monospaced fonts of which there are many out there. Arabic requires ISO-8859-6 as well as Presentation Form-B fonts (without Form-B, Arabic will _NOT_ be usable). It is highly recommended that users search for so-called 'ISO-10646-1' fonts. Do an Internet search or check www.arabeyes.org for further info on where to obtain the necessary Arabic fonts.

Font Installation

- Installation of fonts for X Window systems (Unix/Linux)

  Depending on your system, copy your_ARABIC_FONT file into a directory of your choice. Change to the directory containing the Arabic fonts and execute the following commands:

  ```
  %  mkfontdir
  %  xset +fp path_name_of_arabic_fonts_directory
  ```

Usage

Prior to the actual usage of Arabic within Vim, a number of settings need to be accounted for and invoked.

- Setting the Arabic fonts

  + For Vim GUI set the 'guifont' to your_ARABIC_FONT. This is done by entering the following command in the Vim window.

```
:set guifont=your_ARABIC_FONT
```

NOTE: the string 'your_ARABIC_FONT' is used to denote a complete font name akin to that used in Linux/Unix systems. (e.g. -misc-fixed-medium-r-normal--20-200-75-75-c-100-iso10646-1)

You can append the 'guifont' set command to your vimrc file in order to get the same above noted results. In other words, you can include ':set guifont=your_ARABIC_FONT' to your vimrc file.

+ Under the X Window environment, you can also start Vim with
'-fn your_ARABIC_FONT' option.

o Setting the appropriate character Encoding
To enable the correct Arabic encoding the following command needs
to be appended,

```
:set encoding=utf-8
```

to your vimrc file (entering the command manually into your Vim
window is highly discouraged). In short, include ':set
encoding=utf-8' to your vimrc file.

Attempts to use Arabic without UTF-8 will result the following
warning message,

W17

```
Arabic requires UTF-8, do ':set encoding=utf-8'
```

o Enable Arabic settings [short-cut]

In order to simplify and streamline things, you can either invoke
Vim with the command-line option,

```
% vim -A my_utf8_arabic_file ...
```

or enable 'arabic' via the following command within Vim

```
:set arabic
```

The two above noted possible invocations are the preferred manner
in which users are instructed to proceed. Barring an enabled 'termbidi'
setting, both command options:

1. set the appropriate keymap

2. enable the deletion of a single combined pair character

3. enable rightleft mode

4. enable rightleftcmd mode (affecting the command-line)

5. enable arabicshape mode (do visual character alterations)

You may also append the command to your vimrc file and simply
include ':set arabic' to it.

You are also capable of disabling Arabic support via

```
:set noarabic
```

which resets everything that the command had enabled without touching
the global settings as they could affect other possible open buffers.
In short the 'noarabic' command,

1. resets to the alternate keymap

2. disables the deletion of a single combined pair character

3. disables rightleft mode

NOTE: the 'arabic' command takes into consideration 'termbidi' for
possible external bi-directional (bidi) support from the
terminal ("mlterm" for instance offers such support).
'termbidi', if available, is superior to rightleft support
and its support is preferred due to its level of offerings.
'arabic' when 'termbidi' is enabled only sets the keymap.

For vertical window isolation while setting 'termbidi' an LTR
vertical separator like "l" or "I" may be used. It may also be
hidden by changing its color to the foreground color:

```
:set fillchars=vert:l
:hi WinSeparator ctermbg=White
```

Note that this is a workaround, not a proper solution.

If, on the other hand, you'd like to be verbose and explicit and
are opting not to use the 'arabic' short-cut command, here's what
is needed (i.e. if you use ':set arabic' you can skip this section) -

+ Arabic Keymapping Activation

To activate the Arabic keymap (i.e. to remap your English/Latin
keyboard to look-n-feel like a standard Arabic one), set the
'keymap' option to "arabic". This is done by entering

```
:set keymap=arabic
```

in your VIM window. You can also append the 'keymap' set command to your vimrc file. In other words, you can include ':set keymap=arabic' to your vimrc file.

To turn toggle (or switch) your keymapping between Arabic and the default mapping (English), it is advised that users use the 'CTRL-^' key press while in insert (or add/replace) mode. The command-line will display your current mapping by displaying an "Arabic" string next to your insertion mode (e.g. -- INSERT Arabic --) indicating your current keymap.

+ Arabic deletion of a combined pair character

By default Vim has the 'delcombine' option disabled. This option allows the deletion of ALEF in a LAM_ALEF (LAA) combined character and still retain the LAM (i.e. it reverts to treating the combined character as its natural two characters form -- this also pertains to harakat and their combined forms). You can enable this option by entering

```
:set delcombine
```

in our VIM window. You can also append the 'delcombine' set command to your vimrc file. In other words, you can include ':set delcombine' to your vimrc file.

+ Arabic right-to-left Mode

By default VIM starts in Left-to-right mode. 'rightleft' is the command that allows one to alter a window's orientation - that can be accomplished via,

- Toggling between left-to-right and right-to-left modes is accomplished through ':set rightleft' and ':set norightleft'.
- While in Left-to-right mode, enter ':set rl' in the command line ('rl' is the abbreviation for rightleft).
- Put the ':set rl' line in your vimrc file to start Vim in right-to-left mode permanently.

+ Arabic right-to-left command-line Mode

For certain commands the editing can be done in right-to-left mode. Currently this is only applicable to search commands.

This is controlled with the 'rightleftcmd' option. The default is "search", which means that windows in which 'rightleft' is set will edit search commands in right-left mode. To disable this behavior,

```
:set rightleftcmd=
```

To enable right-left editing of search commands again,

```
:set rightleftcmd&
```

+ Arabic Shaping Mode

To activate the required visual characters alterations (shaping, composing, combining) which the Arabic language requires, enable the 'arabicshape' command. This is done by entering

```
:set arabicshape
```

in our VIM window. You can also append the 'arabicshape' set command to your vimrc file. In other words, you can include ':set arabicshape' to your vimrc file.

Keymap/Keyboard **arabickeymap**

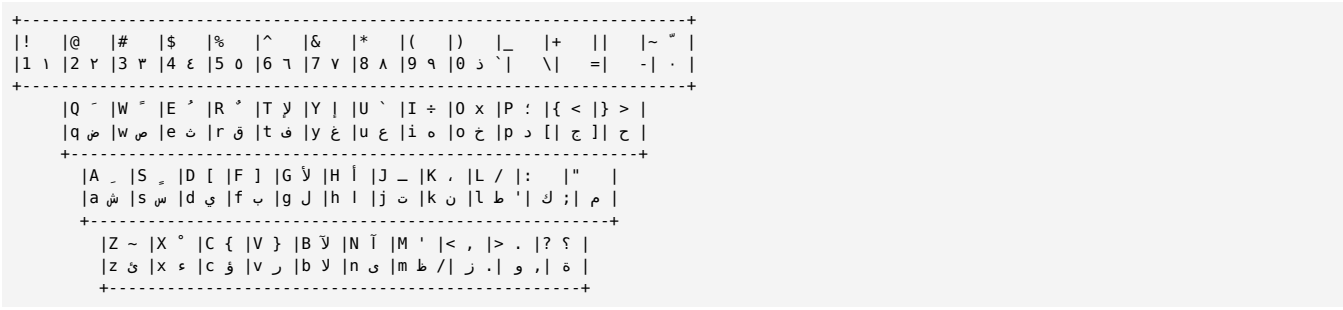
The character/letter encoding used in Vim is the standard UTF-8. It is widely discouraged that any other encoding be used or even attempted.

Note: UTF-8 is an all encompassing encoding and as such is the only supported (and encouraged) encoding with regard to Arabic (all other proprietary encodings should be discouraged and frowned upon).

o Keyboard

+ CTRL-^ in insert/replace mode toggles between Arabic/Latin mode

+ Keyboard mapping is based on the Microsoft's Arabic keymap (the de facto standard in the Arab world):



Restrictions

o Vim in its GUI form does not currently support Bi-directionality
   (i.e. the ability to see both Arabic and Latin intermixed within
   the same line).

Known Bugs

There is one known minor bug,

1. If you insert a haraka (e.g. Fatha (U+064E)) after a LAM (U+0644)

    and then insert an ALEF (U+0627), the appropriate combining will
    not happen due to the sandwiched haraka resulting in something
    that will NOT be displayed correctly.

    WORK-AROUND: Don't include harakats between LAM and ALEF combos.
                 In general, don't anticipate to see correct visual
                 representation with regard to harakats and LAM+ALEF
                 combined characters (even those entered after both
                 characters). The problem noted is strictly a visual
                 one, meaning saving such a file will contain all the
                 appropriate info/encodings - nothing is lost.

No other bugs are known to exist.

Hebrew Language support (options & mapping) for Vim

The supporting 'rightleft' functionality was originally created by Avner Lottem. <alottem at gmail dot com> Ron Aaron <ron at ronware dot org> is currently helping support these features.

Introduction

Hebrew-specific 'keymap' values are "hebrew" and "hebrewp".
Hebrew-useful options are 'delcombine', 'allowrevins', 'revins', 'rightleft' and 'rightleftcmd'.

The 'rightleft' mode reverses the display order, so characters are displayed from right to left instead of the usual left to right. This is useful primarily when editing Hebrew or other Middle-Eastern languages.
See rileft.txt for further details.

Details

+ Options:
  + 'rightleft' ('rl') sets window orientation to right-to-left. This means that the logical text 'ABC' will be displayed as 'CBA', and will start drawing at the right edge of the window, not the left edge.
  + 'keymap' ('kmp') sets keyboard mapping. use values "hebrew" or "hebrewp" (the latter option enables phonetic mapping)
  + 'delcombine' ('deco'), boolean, allows one to remove the niqud or te`amim by pressing 'x' on a character (with associated niqud).
  + 'rightleftcmd' ('rlc') makes the command-prompt for searches show up on the right side. It only takes effect if the window is 'rightleft'.
+ Encoding:
  + Under Unix, ISO 8859-8 encoding (Hebrew letters codes: 224-250).
  + Under MS DOS, PC encoding (Hebrew letters codes: 128-154).
  + You should prefer using UTF8, as it supports the combining-characters ('deco' does nothing if UTF8 encoding is not active).
+ Vim arguments:
  + vim -H file starts editing a Hebrew file, i.e. 'rightleft' is set and 'keymap' is set to "hebrew".
+ Keyboard:
  + The 'allowrevins' option enables the CTRL-_ command in Insert mode.
  + CTRL-_ in Insert mode toggles 'revins'.

    CTRL-_ moves the cursor to the end of the typed text.

    Note: On some keyboards, CTRL-_ is mapped to CTRL-?.
  + Keyboard mapping while 'keymap' is "hebrew" (standard Israeli keyboard):

    q w e r t y u i o p
    / ' ק ר א ט ו ן ם פ

     a s d f g h j k l ; '
     ש ד ג כ ע י ח ל ך ף ,

      z x c v b n m , . /
      ז ס ב ה נ מ צ ת ץ .

    The 'keymap' keyboard can also insert niqud and te`amim. To see what those mappings are, look at the keymap file hebrew.vim etc.

Typing backwards

If the 'revins' (reverse insert) option is set, inserting happens backwards. This can be used to type Hebrew. When inserting characters the cursor is not moved and the text moves rightwards. A <BS> deletes the character under the cursor. CTRL-W and CTRL-U also work in the opposite direction. <BS>, CTRL-W and CTRL-U do not stop at the start of insert or end of line, no matter how the 'backspace' option is set.

There is no reverse replace mode (yet).

If the 'showmode' option is set, "-- REVERSE INSERT --" will be shown in the status line when reverse Insert mode is active.

When the 'allowrevins' option is set, reverse Insert mode can be also entered and exited via CTRL-_.

Pasting when in a rightleft window

When cutting text with the mouse and pasting it in a rightleft window the text will be reversed, because the characters come from the cut buffer from the left to the right, while inserted in the file from the right to the left.   In order to avoid it, toggle 'revins' (by typing CTRL-? or CTRL-_) before pasting.

Russian language localization and support in Vim Russian

1. Introduction russian-intro

Russian language is supported perfectly well in Vim. You can type and view Russian text just as any other, without the need to tweak the settings.

2. Russian keymaps russian-keymap

To switch between languages you can use your system native keyboard switcher, or use one of the Russian keymaps, included in the Vim distribution. For example,

```
:set keymap=russian-jcukenwin
```

In the latter case, you can switch between languages even if you do not have system Russian keyboard or independently from a system-wide keyboard settings. See 'keymap'. You can also map a key to switch between keyboards, if you choose the latter option. See :map.

For your convenience, to avoid switching between keyboards, when you need to enter Normal mode command, you can also set 'langmap' option:

```
:set langmap=ФИСВУАПРШОЛДЬТЩЗЙКЫЕГМЦЧНЯ;ABCDEFGHIJKLMNOPQRSTUVWXYZ,
фисвуапршолдьтщзйкыегмцчня;abcdefghijklmnopqrstuvwxyz
```

You have to type this command in one line, it is wrapped for the sake of readability.

3. Localization russian-l18n

If you wish to use messages, help files, menus and other items translated to Russian, you will need to install the RuVim Language Pack, available in different codepages from

    https://www.sourceforge.net/projects/ruvim/

After downloading an archive from RuVim project, unpack it into your $VIMRUNTIME directory. We recommend using UTF-8 archive.

In order to use the Russian documentation, make sure you have set the 'helplang' option to "ru".

4. Known issues russian-issues

-- If you are using Russian message translations in Win32 console, then you may see the output produced by "vim --help", "vim --version" commands and Win32 console window title appearing in a wrong codepage. This problem is related to a bug in GNU gettext library and may be fixed in the future releases of gettext.

1. Introduction

Vim supports Vietnamese language in the following ways:

- Built-in vietnamese-keymap, which allows you to type Vietnamese characters in Insert-mode and search-commands using US keyboard layout.
- Localization in Vietnamese. See vietnamese-l10n

2. Vietnamese keymaps

vietnamese-keymap

To switch between languages you can use your system native keyboard switcher, or use one of the Vietnamese keymaps included in the Vim distribution, like below

```
:set keymap=vietnamese-telex_utf-8
```

See 'keymap' for more information.

In the latter case, you can type Vietnamese even if you do not have a Vietnamese input method engine (IME) or you want Vim to be independent from a system-wide keyboard settings (when 'imdisable' is set). You can also :map a key to switch between keyboards.

Vim comes with the following Vietnamese keymaps:

- vietnamese-telex_utf-8 Telex input method, UTF-8 encoding.
- vietnamese-viqr_utf-8 VIQR input method, UTF-8 encoding.
- vietnamese-vni_utf-8 VNI input method, UTF-8 encoding.

vietnamese-ime_diff

Since these keymaps were designed to be minimalistic, they do not support all features of the corresponding input methods. The differences are described below:

- You can only type each character individually, entering the base letter first and then the diacritics later. For example, to type the word nếń using vietnamese-vni_utf-8, you must type ne61n, not nen61 or ne6n1
- For characters with more than 1 diacritic, you need to type vowel mark before tone mark. For example, to type ô`using vietnamese-telex_utf-8, you need to type oof, not ofo.
- With vietnamese-telex_utf-8, you need to type all uppercase letters to produce uppercase characters with diacritics. For example, Ừ must be typed as UWF.
- With vietnamese-telex_utf-8, the escape character \ from VNI is added, hence the confusing ooo input to type oo is removed, which could lead to ambiguities. For example, to type the word Đoòng, you would type DDo\ofng.
- Simple Telex (both v1 and v2), including the w[]{} style, is not supported.
- Removing diacritics using z in Telex or 0 in VNI and VIQR is not supported.

3. Localization

vietnamese-l10n

Vim messages are also available in Vietnamese. If you wish to see messages in Vietnamese, you can run the command :language with an argument being the name of the Vietnamese locale. For example,

```
:language vi_VN
```

or

```
:language vi_VN.utf-8
```

Note that the name of the Vietnamese locale may vary depending on your system. See mbyte-first for details.

Nvim UI protocol ui api-ui-events

This document describes the UI protocol. See gui and tui for user-facing
UI components and features.

UI Events

ui-protocol ui-events

UIs can be implemented as external client processes communicating with Nvim
over the RPC API. The default UI model is a terminal-like grid with a single,
monospace font. The UI can opt-in to have windows drawn on separate grids, and
have some elements ("widgets") presented by the UI itself rather than by Nvim
("externalized").

ui-option

Call nvim_ui_attach() to tell Nvim that your program wants to draw the Nvim
screen grid with a size of width × height cells. This is typically done by an
embedder at startup (see ui-startup), but UIs can also connect to a running
Nvim instance and invoke nvim_ui_attach(). The options parameter is a map
with these (optional) keys:

ui-rgb

- rgb Decides the color format.
  - true: (default) 24-bit RGB colors
  - false: Terminal colors (8-bit, max 256)

ui-override

- override Decides how UI capabilities are resolved.
  - true: Enable requested UI capabilities, even if not
    supported by all connected UIs (including TUI).
  - false: (default) Disable UI capabilities not
    supported by all connected UIs (including TUI).

ui-ext-options

- ext_cmdline Externalize the cmdline. ui-cmdline
- ext_hlstate Detailed highlight state. ui-hlstate
- ext_linegrid Sets ext_linegrid implicitly.
  Line-based grid events. ui-linegrid
  Deactivates ui-grid-old implicitly.
- ext_messages Externalize messages. ui-messages
  Sets ext_linegrid and ext_cmdline implicitly.
- ext_multigrid Per-window grid events. ui-multigrid
  Sets ext_linegrid implicitly.
- ext_popupmenu Externalize popupmenu-completion and
  'wildmenu'. ui-popupmenu
- ext_tabline Externalize the tabline. ui-tabline
- ext_termcolors Use external default colors.
- term_name Sets the name of the terminal 'term'.
- term_colors Sets the number of supported colors 't_Co'.
- stdin_fd Treat this fd as if it were stdin when using --.
  Only from --embed UI on startup. ui-startup-stdin
- stdin_tty Tells if stdin is a tty or not.

- stdout_tty Tells if stdout is a tty or not.

Specifying an unknown option is an error; UIs can check the api-metadata ui_options key for supported options.

By default Nvim requires all connected UIs to support the same capabilities, thus the active capabilities are the intersection of those requested. UIs may specify ui-override to invert this behavior (useful for debugging). The "option_set" event announces which capabilities are active.

Nvim sends RPC notifications to all attached UIs, with method name "redraw" and a single argument: an array (batch) of screen "update events". Each update event is itself an array whose first element is the event name and remaining elements are event-parameter tuples. Thus multiple events of the same kind can be sent contiguously without repeating the event name.

Example of a typical "redraw" batch in a single RPC notification:

```
['notification', 'redraw',
  [
    ['grid_resize', [2, 77, 36]],
    ['grid_line',
      [2, 0, 0, [[' ' , 0, 77]], false],
      [2, 1, 0, [['~', 7], [' ', 7, 76]], false],
      [2, 9, 0, [['~', 7], [' ', 7, 76]], false],
      ...
      [2, 35, 0, [['~', 7], [' ', 7, 76]], false],
      [1, 36, 0, [['[', 9], ['N'], ['o'], [' '], ['N'], ['a'], ['m'], ['e'], [']']], false],
      [1, 36, 9, [[' ', 9, 50]], false],
      [1, 36, 59, [['0', 9], [','], ['0'], ['-' ], ['1'], [' ', 9, 10], ['A'], ['l', 9, 2]], false]
    ],
    ['msg_showmode', [[]]],
    ['win_pos', [2, 1000, 0, 0, 77, 36]],
    ['grid_cursor_goto', [2, 0, 0]],
    ['flush', []]
  ]
]
```

Events must be handled in-order. Nvim sends a "flush" event when it has completed a redraw of the entire screen (so all windows have a consistent view of buffer state, options, etc.). Multiple "redraw" batches may be sent before the entire screen has been redrawn, with "flush" following only the last batch. The user should only see the final state (when "flush" is sent), not any intermediate state while processing part of the batch array, nor after a batch not ending with "flush".

By default, Nvim sends ui-global and ui-grid-old events (for backwards compatibility); these suffice to implement a terminal-like interface. However the new ui-linegrid represents text more efficiently (especially highlighted text), and allows UI capabilities requiring multiple grids. New UIs should implement ui-linegrid instead of ui-grid-old.

Nvim optionally sends various screen elements "semantically" as structured events instead of raw grid-lines, as specified by ui-ext-options. The UI must present such elements itself, Nvim will not draw them on the grid.

Future versions of Nvim may add new update kinds and may append new parameters to existing update kinds. Clients must be prepared to ignore such extensions, for forward-compatibility. api-contract

UI startup ui-startup

UI embedders (clients that start Nvim with --embed and later call nvim_ui_attach()) must start Nvim without --headless:

```
nvim --embed
```

Nvim will pause before loading startup files and reading buffers, so the UI has a chance to invoke requests and do early initialization. Startup will continue when the UI invokes nvim_ui_attach().

A basic UI only needs to call nvim_ui_attach() and then prepare to handle UI events. A more featureful UI, should use the following startup procedure:

1. Invoke nvim_get_api_info(), if needed to setup the client library and/or

   to get the list of supported UI extensions.
2. Do any configuration that should happen before user config is loaded.

   Buffers and windows are not available at this point, but this could be used to set g: variables visible to init.vim
3. If the UI wants to do additional setup after user config is loaded,

   register a VimEnter autocmd:

```
nvim_command("autocmd VimEnter * call rpcrequest(1, 'vimenter')")
```

4. Now invoke nvim_ui_attach(). The UI must handle user input by now:

   sourcing init.vim and loading buffers might lead to blocking prompts.
5. If step 3 was used, Nvim will send a blocking "vimenter" request to the UI.

   Inside this request handler, the UI can safely do any initialization before entering normal mode, for example reading variables set by init.vim.

UIs should also handle ui-event-restart.

ui-startup-stdin

UIs can support reading from stdin (like command | nvim -, see --) as follows:

1. The embedding process detects that the stdin fd is not a terminal.

2. It then needs to forward this fd to Nvim. Because fd=0 is already is used

   to send RPC data from embedder to Nvim, it must use some other file descriptor, like fd=3 or higher.
3. Then pass the fd as the stdin_fd parameter of nvim_ui_attach. Nvim will

   read it as text into buffer 1.

Global Events ui-global

The following UI events are always emitted, and describe global state of the editor.

["set_title", title]
["set_icon", icon]
Set the window title, and icon (minimized) window title, respectively. In windowing systems not distinguishing between the two, "set_icon" can be ignored.

["mode_info_set", cursor_style_enabled, mode_info]
cursor_style_enabled is a boolean indicating if the UI should set the cursor style. mode_info is a list of mode property maps. The current mode is given by the mode_idx field of the mode_change event.

Each mode property map may contain these keys:

KEY	DESCRIPTION
cursor_shape:	"block", "horizontal", "vertical" cell_percentage: Cell % occupied by the cursor. blinkwait, blinkon, blinkoff: See cursor-blinking.
attr_id:	Cursor attribute id (defined by hl_attr_define). When attr_id is 0, the background and foreground colors should be swapped.
attr_id_lm:	Cursor attribute id for when :lmap is on.
short_name:	Mode code name, see 'guicursor'.
name:	Mode descriptive name.
mouse_shape:	(To be implemented.)

Some keys are missing in some modes.

The following keys are deprecated:

hl_id: Use attr_id instead.
id_lm: Use attr_id_lm instead.

["option_set", name, value]
UI-related option changed, where name is one of:

- 'arabicshape'
- 'ambiwidth'
- 'emoji'
- 'guifont'
- 'guifontwide'
- 'linespace'
- 'mousefocus'
- 'mousehide'

- 'mousemoveevent'
- 'pumblend'
- 'showtabline'
- 'termguicolors'
- "ext_*" (all ui-ext-options)

Triggered when the UI first connects to Nvim, and whenever an option is changed by the user or a plugin.

Options are not represented here if their effects are communicated in other UI events. For example, instead of forwarding the 'mouse' option value, the "mouse_on" and "mouse_off" UI events directly indicate if mouse support is active. Some options like 'ambiwidth' have already taken effect on the grid, where appropriate empty cells are added, however a UI might still use such options when rendering raw text sent from Nvim, like for ui-cmdline.

["chdir", path]
The current-directory changed to path.

["mode_change", mode, mode_idx]
Editor mode changed. The mode parameter is a string representing the current mode. mode_idx is an index into the array emitted in the mode_info_set event. UIs should change the cursor style according to the properties specified in the corresponding item. The set of modes reported will change in new versions of Nvim, for instance more submodes and temporary states might be represented as separate modes.

["mouse_on"]
["mouse_off"]
'mouse' was enabled/disabled in the current editor mode. Useful for a terminal UI, or embedding into an application where Nvim mouse would conflict with other usages of the mouse. Other UI:s may ignore this event.

["busy_start"]
["busy_stop"]
Indicates to the UI that it must stop rendering the cursor. This event is misnamed and does not actually have anything to do with busyness.

["connect", server_addr]
User invoked the :connect command. Nvim detached the current UI.

UI is expected to:
1. Attach to the server at server_addr.

ui-event-restart

["restart", listen_addr, chan_config]
User invoked the :restart command. Nvim started a new server. The old server is about to exit and close its channel. chan_config gives the channel-specific properties from the old server, which the UI should pass to nvim_chan_set() intact.

UI is expected to:
1. Wait for the channel to be closed, to confirm that the old server actually exited.
2. Call nvim_chan_set(0, chan_config) on the new server at listen_addr, to restore the channel properties (e.g. :detach!).
3. Attach to the new server at listen_addr.

["suspend"]
:suspend command or CTRL-Z mapping is used. A terminal client (or another client where it makes sense) could suspend itself. Other clients can safely ignore it.

["update_menu"]
The menu mappings changed.

["bell"]
["visual_bell"]
Notify the user with an audible or visual bell, respectively.

["flush"]
Nvim is done redrawing the screen. For an implementation that renders to an internal buffer, this is the time to display the redrawn parts to the user.

["ui_send", content]

Write {content} to the connected TTY. Only UIs that have the
"stdout_tty" ui-option set will receive this event.

Grid Events (line-based) ui-linegrid

Activated by the ext_linegrid ui-option. Recommended for all new UIs.
Deactivates ui-grid-old implicitly.

Unlike ui-grid-old, this UI extension emits a single grid_line event to
update a screen-line (whereas the old protocol emitted separate cursor,
highlight and text events per screen-line).

Most of these events take a grid index as first parameter. Grid 1 is the
global grid used by default for the entire editor screen state. The
ext_linegrid capability by itself will never cause any additional grids to
be created; to enable per-window grids, activate ui-multigrid.

Highlight attribute groups are predefined. UIs should maintain a table to map
numerical highlight ids to the actual attributes.

["grid_resize", grid, width, height]
Resize a grid. If grid wasn't seen by the client before, a new grid is
being created with this size.

["default_colors_set", rgb_fg, rgb_bg, rgb_sp, cterm_fg, cterm_bg]
The first three arguments set the default foreground, background and
special colors respectively. cterm_fg and cterm_bg specifies the
default color codes to use in a 256-color terminal.

The RGB values will always be valid colors, by default. If no
colors have been set, they will default to black and white, depending
on 'background'. By setting the ext_termcolors option, instead
-1 will be used for unset colors. This is mostly useful for a TUI
implementation, where using the terminal builtin ("ANSI") defaults
are expected.

Note: Unlike the corresponding ui-grid-old events, the screen is not
always cleared after sending this event. The UI must repaint the
screen with changed background color itself.

ui-event-hl_attr_define

["hl_attr_define", id, rgb_attr, cterm_attr, info]
Add a highlight with id to the highlight table, with the
attributes specified by the rgb_attr and cterm_attr dicts, with the
following (all optional) keys.

foreground:	foreground color.
background:	background color.
special:	color to use for various underlines, when present.
reverse:	reverse video. Foreground and background colors are switched.
italic:	italic text.
bold:	bold text.
strikethrough:	struckthrough text.
underline:	underlined text. The line has special color.
undercurl:	undercurled text. The curl has special color.
underdouble:	double underlined text. The lines have special color.
underdotted:	underdotted text. The dots have special color.
underdashed:	underdashed text. The dashes have special color.
altfont:	alternative font.
dim:	half-bright/faint text.
blink:	blinking text.
conceal:	concealed/hidden text.
overline:	overlined text.
blend:	blend level (0-100). Could be used by UIs to support blending floating windows to the background or to signal a transparent cursor.
url:	URL associated with this highlight. UIs should present the region as a clickable hyperlink.

For absent color keys the default color should be used. Don't store
the default value in the table, rather a sentinel value, so that
a changed default color will take effect.
All boolean keys default to false, and will only be sent when they
are true.

Highlights are always transmitted both for both the RGB format and as
terminal 256-color codes, as the rgb_attr and cterm_attr parameters
respectively. The ui-rgb option has no effect anymore.
Most external UIs will only need to store and use the rgb_attr
attributes.

id 0 will always be used for the default highlight with colors defined
by default_colors_set and no styles applied.

Note: Nvim may reuse id values if its internal highlight table is full.
In that case Nvim will always issue redraws of screen cells that are
affected by redefined ids, so UIs do not need to keep track of this
themselves.

info is an empty array unless ui-hlstate is enabled.

["hl_group_set", name, hl_id]
The built-in highlight group name was set to use the attributes hl_id
defined by a previous hl_attr_define call. This event is not needed
to render the grids which use attribute ids directly, but is useful

for a UI who want to render its own elements with consistent highlighting. For instance a UI using ui-popupmenu events, might use the hl-Pmenu family of builtin highlights.

ui-event-grid_line

["grid_line", grid, row, col_start, cells, wrap]
Redraw a continuous part of a row on a grid, starting at the column col_start. cells is an array of arrays each with 1 to 3 items: [text(, hl_id, repeat)] . text is the UTF-8 text that should be put in a cell, with the highlight hl_id defined by a previous hl_attr_define call. If hl_id is not present the most recently seen hl_id in the same call should be used (it is always sent for the first cell in the event). If repeat is present, the cell should be repeated repeat times (including the first time), otherwise just once.

The right cell of a double-width char will be represented as the empty string. Double-width chars never use repeat.

If the array of cell changes doesn't reach to the end of the line, the rest should remain unchanged. A whitespace char, repeated enough to cover the remaining line, will be sent when the rest of the line should be cleared.

wrap is a boolean indicating that this line wraps to the next row. When redrawing a line which wraps to the next row, Nvim will emit a grid_line event covering the last column of the line with wrap set to true, followed immediately by a grid_line event starting at the first column of the next row.

["grid_clear", grid]
Clear a grid.

["grid_destroy", grid]
grid will not be used anymore and the UI can free any data associated with it.

["grid_cursor_goto", grid, row, col]
Makes grid the current grid and row, col the cursor position on this grid. This event will be sent at most once in a redraw batch and indicates the visible cursor position.

["grid_scroll", grid, top, bot, left, right, rows, cols]
Scroll a region of grid. This is semantically unrelated to editor scrolling, rather this is an optimized way to say "copy these screen cells".

The following diagrams show what happens per scroll direction.
"===" represents the SR (scroll region) boundaries.
"---" represents the moved rectangles.
Note that dst and src share a common region.

If rows is bigger than 0, move a rectangle in the SR up, this can happen while scrolling down.

```
+-------------------------+
| (clipped above SR)      |            ^
|=========================| dst_top    |
| dst (still in SR)       |            |
+-------------------------+ src_top    |
| src (moved up) and dst  |            |
|-------------------------| dst_bot    |
| src (invalid)           |            |
+=========================+ src_bot
```

If rows is less than zero, move a rectangle in the SR down, this can happen while scrolling up.

```
+=========================+ src_top
| src (invalid)           |            |
|------------------------ | dst_top    |
| src (moved down) and dst|            |
+-------------------------+ src_bot    |
| dst (still in SR)       |            |
|=========================| dst_bot    |
| (clipped below SR)      |            v
+-------------------------+
```

cols is always zero in this version of Nvim, and reserved for future use.

Note when updating code from ui-grid-old events: ranges are end-exclusive, which is consistent with API conventions, but different from set_scroll_region which was end-inclusive.

The scrolled-in area will be filled using ui-event-grid_line directly after the scroll event. The UI thus doesn't need to clear this area as part of handling the scroll event.

Grid Events (cell-based)

ui-grid-old

This is the legacy representation of the screen grid, emitted if ui-linegrid is not active. New UIs should implement ui-linegrid instead.

["resize", width, height]
The grid is resized to width and height cells.

["clear"]
Clear the grid.

["eol_clear"]
Clear from the cursor position to the end of the current line.

["cursor_goto", row, col]
Move the cursor to position (row, col). Currently, the same cursor is used to define the position for text insertion and the visible cursor. However, only the last cursor position, after processing the entire array in the "redraw" event, is intended to be a visible cursor position.

["update_fg", color]
["update_bg", color]
["update_sp", color]
Set the default foreground, background and special colors respectively.

ui-event-highlight_set

["highlight_set", attrs]
Set the attributes that the next text put on the grid will have. attrs is a dict with the keys below. Any absent key is reset to its default value. Color defaults are set by the update_fg etc updates. All boolean keys default to false.

foreground: foreground color.
background: background color.
special: color to use for various underlines, when present.
reverse: reverse video. Foreground and background colors are switched.
italic: italic text.
bold: bold text.
strikethrough: struckthrough text.
underline: underlined text. The line has special color.
undercurl: undercurled text. The curl has special color.
underdouble: double underlined text. The lines have special color.
underdotted: underdotted text. The dots have special color.
underdashed: underdashed text. The dashes have special color.
dim: half-bright/faint text.
blink: blinking text.
conceal: concealed/hidden text.
overline: overlined text.

["put", text]
The (utf-8 encoded) string text is put at the cursor position (and the cursor is advanced), with the highlights as set by the last highlight_set update.

["set_scroll_region", top, bot, left, right]
Define the scroll region used by scroll below.

Note: ranges are end-inclusive, which is inconsistent with API conventions.

["scroll", count]
Scroll the text in the scroll region. The diagrams below illustrate what will happen, depending on the scroll direction. "=" is used to represent the SR(scroll region) boundaries and "-" the moved rectangles. Note that dst and src share a common region.

If count is bigger than 0, move a rectangle in the SR up, this can happen while scrolling down.

```
+-------------------------+
| (clipped above SR)      |            ^
|=========================| dst_top    |
| dst (still in SR)       |            |
+-------------------------+ src_top    |
| src (moved up) and dst  |            |
|-------------------------| dst_bot    |
| src (cleared)           |            |
+=========================+ src_bot
```

If count is less than zero, move a rectangle in the SR down, this can happen while scrolling up.

```
+=========================+ src_top
| src (cleared)           |            |
|------------------------ | dst_top    |
| src (moved down) and dst|            |
+-------------------------+ src_bot    |
| dst (still in SR)       |            |
|=========================| dst_bot    |
| (clipped below SR)      |            v
+-------------------------+
```

Highlight Events

ui-hlstate

Activated by the ext_hlstate ui-option.
Activates ui-linegrid implicitly.

If ext_hlstate is enabled, Nvim will emit detailed highlight state in ui-linegrid events. Otherwise (by default) Nvim only describes grid cells

using the final calculated highlight attributes described at ui-event-highlight_set.

ext_hlstate provides a semantic description of active highlights for each grid cell. Highlights are predefined in a table, see ui-event-hl_attr_define and ui-event-grid_line.

The info parameter in hl_attr_define contains a semantic description of the highlights. Because highlight groups can be combined, this is an array where the highest-priority item is last. Each item is a dict with these keys:

```
    kind:       always present. One of the following values:
        "ui":       Builtin UI highlight. highlight-groups
        "syntax":   Highlight applied to a buffer by a syntax declaration or
                    other runtime/plugin functionality such as
                    nvim_buf_set_extmark()
        "terminal": highlight from a process running in a terminal-emulator.
                    Contains no further semantic information.
    ui_name:    Highlight name from highlight-groups. Only for "ui" kind.
    hi_name:    Name of the final :highlight group where the used
                attributes are defined.
    id: Unique numeric id representing this item.
```

Note: "ui" items will have both ui_name and hi_name present. These can differ, because the builtin group was linked to another group :hi-link , or because 'winhighlight' was used. UI items will be transmitted, even if the highlight group is cleared, so ui_name can always be used to reliably identify screen elements, even if no attributes have been applied.

Multigrid Events ui-multigrid

Activated by the ext_multigrid ui-option.
Activates ui-linegrid implicitly.

See ui-linegrid for grid events.
See nvim_ui_try_resize_grid() to request changing the grid size.
See nvim_input_mouse() for sending mouse events to Nvim.

The multigrid extension gives UIs more control over how windows are displayed:

- UIs receive updates on a separate grid for each window.
- UIs can set the grid size independently of how much space the window occupies on the global layout. So the UI could use a different font size per-window. Or reserve space around the border of the window for its own elements, such as scrollbars from the UI toolkit.
- A dedicated grid is used for messages, which may scroll over the window area. (Alternatively ui-messages can be used).

By default, the grid size is handled by Nvim and set to the outer grid size (i.e. the size of the window frame in Nvim) whenever the split is created. Once a UI sets a grid size, Nvim does not handle the size for that grid and the UI must change the grid size whenever the outer size is changed. To delegate grid-size handling back to Nvim, request the size (0, 0).

A window can be hidden and redisplayed without its grid being deallocated. This can happen multiple times for the same window, for instance when switching tabs.

["win_pos", grid, win, start_row, start_col, width, height]
Set the position and size of the grid in Nvim (i.e. the outer grid size). If the window was previously hidden, it should now be shown again.

["win_float_pos", grid, win, anchor, anchor_grid, anchor_row, anchor_col, mouse_enabled, zindex, compindex, screen_row, screen_col]
Display or reconfigure floating window win.

There are two alternative ways of positioning the window

- Manually - The window should be displayed above another grid anchor_grid at the specified position anchor_row and anchor_col. For the meaning of anchor and more details of positioning, see nvim_open_win(). **NOTE:** you have to manually ensure that the window fits the screen, possibly by further reposition it. Ignore screen_row and screen_col in this case.
- Let nvim take care of the positioning - You can ignore anchor and display the window at screen_row and screen_col.

mouse_enabled is true if the window can receive mouse events.

zindex is the configured zindex, while compindex is the exact rendering order of the windows determined by nvim. To render exactly like the TUI, first render all the non-floating windows, then render in the compindex order, overwriting any floating window cells. Finally, blend the floating window cells against the non-floating background. To add more blending, you can group the windows by zindex, and blend between the layers. But note that windows inside the same zindex should still overwrite previous cells inside the same layer without blending. This ensures that plugins that render multiple windows, to add borders for example, work as expected.

["win_external_pos", grid, win]
Display or reconfigure external window win. The window should be displayed as a separate top-level window in the desktop environment, or something similar.

["win_hide", grid]
Stop displaying the window. The window can be shown again later.

["win_close", grid]
Close the window.

["msg_set_pos", grid, row, scrolled, sep_char, zindex, compindex]
Display messages on grid. The grid will be displayed at row on the default grid (grid=1), covering the full column width. scrolled indicates whether the message area has been scrolled to cover other grids. It can be useful to draw a separator then msgsep. The Builtin TUI draws a full line filled with sep_char ('fillchars' msgsep field) and hl-MsgSeparator highlight.

When ui-messages is active, no message grid is used, and this event will not be sent.

zindex and compindex have the same meaning as for win_float_pos. The zindex always has a fixed value of 200 and included for completeness.

["win_viewport", grid, win, topline, botline, curline, curcol, line_count, scroll_delta]
Indicates the range of buffer text displayed in the window, as well as the cursor position in the buffer. All positions are zero-based. botline is set to one more than the line count of the buffer, if there are filler lines past the end. scroll_delta contains how much the top line of a window moved since win_viewport was last emitted. It is intended to be used to implement smooth scrolling. For this purpose it only counts "virtual" or "displayed" lines, so folds only count as one line. When scrolling more than a full screen it is an approximate value.

All updates, such as grid_line, in a batch affects the new viewport, despite the fact that win_viewport is received after the updates. Applications implementing, for example, smooth scrolling should take this into account and keep the grid separated from what's displayed on the screen and copy it to the viewport destination once win_viewport is received.

["win_viewport_margins", grid, win, top, bottom, left, right]
Indicates the margins of a window grid which are _not_ part of the viewport as indicated by the win_viewport event. This happens e.g. in the presence of 'winbar' and floating window borders.

["win_extmark", grid, win, ns_id, mark_id, row, col]
Updates the position of an extmark which is currently visible in a window. Only emitted if the mark has the ui_watched attribute.

Popupmenu Events

ui-popupmenu

Activated by the ext_popupmenu ui-option.

This UI extension delegates presentation of the popupmenu-completion and command-line 'wildmenu'.

The UI decides how to present the menu. For example, depending on the last mode_change event, command-line wildmenu may be presented horizontally, while insert-mode completion would show a vertical popupmenu.

["popupmenu_show", items, selected, row, col, grid]
Show popupmenu-completion. items is an array of completion items to show; each item is an array of the form [word, kind, menu, info] as defined at complete-items, except that word is replaced by abbr if present. selected is the initially-selected item, a zero-based index into the array of items (-1 if no item is selected). row and col give the anchor position, where the first character of the completed word will be. When ui-multigrid is used, grid is the grid for the anchor position. When ext_cmdline is active, grid is set to -1 to indicate the popupmenu should be anchored to the external cmdline. Then col will be a byte position in the cmdline text.

["popupmenu_select", selected]
Select an item in the current popupmenu. selected is a zero-based index into the array of items from the last popupmenu_show event, or -1 if no item is selected.

["popupmenu_hide"]
Hide the popupmenu.

Tabline Events

ui-tabline

Activated by the ext_tabline ui-option.

["tabline_update", curtab, tabs, curbuf, buffers]
Tabline was updated. UIs should present this data in a custom tabline widget. **Note:** options curbuf + buffers were added in API7.
curtab: Current Tabpage
tabs: List of Dicts [{ "tab": Tabpage, "name": String }, ...]
curbuf: Current buffer handle.
buffers: List of Dicts [{ "buffer": buffer handle, "name": String}, ...]

Cmdline Events

ui-cmdline

Activated by the ext_cmdline ui-option.

This UI extension delegates presentation of the cmdline (except 'wildmenu').
For command-line 'wildmenu' UI events, activate ui-popupmenu.

["cmdline_show", content, pos, firstc, prompt, indent, level, hl_id]
Triggered when the cmdline is displayed or changed.

content
Array of [attr_id, text_chunk, hl_id] tuples, building up the cmdline text of chunks with different highlights.

pos
Position of the cursor in the cmdline (byte offset).

firstc
Character preceding the cmdline content (getcmdtype()).

prompt
String preceding the cmdline content when an input() prompt is active, false otherwise. Highlighted with hl_id.

indent
Number of spaces the content should be indented.

level
Number indicating the cmdline recursion level. Incremented by e.g c_CTRL-R_=, prompts or a commandline invoked from the cmdline-window.

hl_id
Highlight group ID of the prompt, -1 with no active prompt.

["cmdline_pos", pos, level]
Change the cursor position in the cmdline.

["cmdline_special_char", c, shift, level]
Display a special char in the cmdline at the cursor position. This is typically used to indicate a pending state, e.g. after c_CTRL-V. If shift is true the text after the cursor should be shifted, otherwise it should overwrite the char at the cursor.

Should be hidden at next cmdline_show.

["cmdline_hide", level, abort]
Hide the cmdline. level is the nesting level of the cmdline being hidden. abort is true if the cmdline is hidden after an aborting condition (c_Esc or c_CTRL-C).

["cmdline_block_show", lines]
Show a block of context to the current command line. For example if the user defines a :function interactively:

```
:function Foo()
:  echo "foo"
:
```

lines is a list of lines of highlighted chunks, in the same form as the "cmdline_show" contents parameter.

["cmdline_block_append", line]
Append a line at the end of the currently shown block.

["cmdline_block_hide"]
Hide the block.

Message/Dialog Events

ui-messages

Activated by the ext_messages ui-option.
Activates ui-linegrid and ui-cmdline implicitly.

This UI extension delegates presentation of messages and dialogs. Messages that would otherwise render in the message/cmdline screen space, are emitted as UI events.

Nvim will not allocate screen space for the cmdline or messages. 'cmdheight' will be set to zero, but can be changed and used for the replacing cmdline or message window. Cmdline state is emitted as ui-cmdline events, which the UI must handle.

["msg_show", kind, content, replace_last, history, append, id, trigger]
Display a message to the user. Update (replace) any existing message matching id.

kind
Name indicating the message kind:

"" (empty)	Unknown (consider a feature-request)
"empty"	Empty message (:echo ""), with empty content. Should clear messages sharing the 'cmdheight' area if it is the only message in a batch.
"confirm"	Message preceding a prompt (:confirm, confirm(), inputlist(), z=, …)
"emsg"	Error (errors, internal error, :throw, …)
"echo"	:echo message
"echomsg"	:echomsg message
"echoerr"	:echoerr message
"completion"	ins-completion-menu message
"list_cmd"	List output for various commands (:ls, :set, …)

"lua_error"   Error in :lua code
"lua_print"   print() from :lua code
"progress"   Progress message emitted by nvim_echo().
                Internal progress messages use these ids:

- nvim.bufwrite "<filename>"

- nvim.completion

- nvim.indent

"quickfix"   Quickfix navigation message
"rpc_error"   Error response from rpcrequest()
"search_cmd"   Entered search command
"search_count" Search count message ("S" flag of 'shortmess')
"shell_cmd"   :!cmd executed command
"shell_err"   :!cmd shell stderr output
"shell_out"   :!cmd shell stdout output
"shell_ret"   :!cmd shell return code
"undo"   :undo and :redo message
"verbose"   'verbose' message
"wildlist"   'wildmode' "list" message
"wmsg"   Warning ("search hit BOTTOM", W10, …)

New kinds may be added in the future; clients should treat unknown kinds as the empty kind.

content
Array of [attr_id, text_chunk, hl_id] tuples, building up the message text of chunks with different highlights. No extra spacing should be added between chunks, the text_chunk by itself contains any necessary whitespace. Messages can contain line breaks "\n".

replace_last
Decides how multiple messages should be displayed:
false: Display the message together with all previous messages that are still visible.
true: Replace the message in the most-recent msg_show call, but any other visible message should still remain.

history
True if the message was added to the :messages history.

append
True if the message should be appended to the previous message, rather than started on a new line. Is set for :echon.

id
Unique identifier for the message. It can either be an integer or string. A (visible) message with the same id should be replaced.

trigger
Type of action that triggered the message:
"" (empty)   Unknown (consider a feature-request)
"typed_cmd"   Interactively typed command on the cmdline

["msg_clear"]
Clear all messages currently displayed by "msg_show", emitted after clearing the screen (messages sent by other "msg_" events below should not be affected).

Guidance: The "clear messages" behavior is UI-specific. If the UI presents messages in a new window, it may choose to clear messages after a few seconds. If the UI presents messages in a persistent area (e.g. cmdline), it should clear messages at the start of the next batch (typically, the next event-loop cycle).

["msg_showmode", content]
Shows 'showmode' and recording messages. content has the same format as in "msg_show". This event is sent with empty content to hide the last message.

["msg_showcmd", content]
Shows 'showcmd' messages. content has the same format as in "msg_show". This event is sent with empty content to hide the last message.

["msg_ruler", content]
Used to display 'ruler' when there is no space for the ruler in a statusline. content has the same format as in "msg_show". This event is sent with empty content to hide the last message.

["msg_history_show", entries, prev_cmd]
Sent when :messages or g< command is invoked. History is sent as a list of entries, where each entry is a [kind, content, append] tuple.

prev_cmd
True when sent with g< command, false with :messages.

Nvim API api

Nvim exposes a powerful API that can be used by plugins and external processes via RPC, Lua and Vimscript (eval-api).

Applications can also embed libnvim to work with the C API directly.

API Usage

api-rpc RPC rpc

msgpack-rpc

RPC is the main way to control Nvim programmatically. Nvim implements the MessagePack-RPC protocol with these extra (out-of-spec) constraints:

1. Responses must be given in reverse order of requests (like "unwinding a stack").
2. Nvim processes all messages (requests and notifications) in the order they are received.

MessagePack-RPC specification: https://github.com/msgpack-rpc/msgpack-rpc/blob/master/spec.md
https://github.com/msgpack/msgpack/blob/0b8f5ac/spec.md

Many clients use the API: user interfaces (GUIs), remote plugins, scripts like "nvr" (https://github.com/mhinz/neovim-remote). Even Nvim itself can control other Nvim instances. API clients can:

- Call any API function
- Listen for events
- Receive remote calls from Nvim

The RPC API is like a more powerful version of Vim's "clientserver" feature.

CONNECTING

rpc-connecting

See channel-intro for various ways to open a channel. Channel-opening functions take an `rpc` key in the options dict. RPC channels can also be opened by other processes connecting to TCP/IP sockets or named pipes listened to by Nvim.

Nvim creates a default RPC socket at startup, given by v:servername. To start with a TCP/IP socket instead, use --listen with a TCP-style address:

```
nvim --listen 127.0.0.1:6666
```

More endpoints can be started with serverstart().

Note that localhost TCP sockets are generally less secure than named pipes, and can lead to vulnerabilities like remote code execution.

Connecting to the socket is the easiest way a programmer can test the API, which can be done through any msgpack-rpc client library or full-featured api-client. Here's a Ruby script that prints "hello world!" in the current Nvim instance:

```
#!/usr/bin/env ruby
# Requires msgpack: gem install msgpack
#
# To run this script, use Nvim's built-in terminal emulator:
#
#   :term ruby hello.rb
#
# Or from another shell by setting NVIM:
# :echo v:servername -- this is the unix socket path address
# $ NVIM=[unix_socket_path_address] ruby ./hello.rb
# or if you run the command above: nvim --listen 127.0.0.1:6666 -- tcp socket address
# $ NVIM=127.0.0.1:6666 ruby hello.rb

require 'msgpack'
require 'socket'

sock =
if ENV['NVIM'] =~ /:\d+$/
    host, port = ENV['NVIM'].split(':')
    TCPSocket.new(host, port)
else
    UNIXSocket.new(ENV['NVIM'])
end
sock.write([0, 0, 'nvim_command', ['echo "hello world!"']].to_msgpack)
p MessagePack::Unpacker.new(sock).read.last
```

Another way is to use the Python REPL with the "pynvim" package, where API functions can be called interactively (requires `pip install pynvim` or `uv add pynvim`):

```
>>> from pynvim import attach
>>> nvim = attach('socket', path='[address]')
>>> nvim.command('echo "hello world!"')
```

You can also embed Nvim via jobstart(), and communicate using rpcrequest() and rpcnotify():

```
let nvim = jobstart(['nvim', '--embed'], {'rpc': v:true})
echo rpcrequest(nvim, 'nvim_eval', '"Hello " . "world!"')
call jobstop(nvim)
```

API Definitions

api-definitions

api-types

The Nvim C API defines custom types for all function parameters. Some are just typedefs around C99 standard types, others are Nvim-defined

data structures.

Basic types

API Type	C type
Nil	
Boolean	bool
Integer (signed 64-bit integer)	int64_t
Float (IEEE 754 double precision)	double
String	{char* data, size_t size} struct
Array	kvec
Dict (msgpack: map)	kvec
Object	any of the above

Note:
- Empty Array is accepted as a valid Dictionary parameter.
- Functions cannot cross RPC boundaries. But API functions (e.g. nvim_create_autocmd()) may support Lua function parameters for non-RPC invocations.

Special types (msgpack EXT)

These are integer typedefs discriminated as separate Object subtypes. They can be treated as opaque integers, but are mutually incompatible: Buffer may be passed as an integer but not as Window or Tabpage.

The EXT object data is the (integer) object handle. The EXT type codes given in the api-metadata `types` key are stable: they will not change and are thus forward-compatible.

EXT Type	C type	Data
Buffer	enum value kObjectTypeBuffer	bufnr()
Window	enum value kObjectTypeWindow	window-ID
Tabpage	enum value kObjectTypeTabpage	internal handle

api-indexing

Most of the API uses 0-based indices, and ranges are end-exclusive. For the end of a range, -1 denotes the last line/column. A column in buffer text is a byte index, not a character index or screen cell. See vim.pos, vim.range for representing and converting positions.

Exception: the following API functions use "mark-like" indexing (1-based lines, 0-based columns, see also vim.pos.mark()):

- nvim_get_mark()
- nvim_buf_get_mark()
- nvim_buf_set_mark()
- nvim_win_get_cursor()
- nvim_win_set_cursor()

Exception: the following API functions use extmarks indexing (0-based indices, end-inclusive, see also vim.pos.extmark()):

- nvim_buf_del_extmark()
- nvim_buf_get_extmark_by_id()
- nvim_buf_get_extmarks()
- nvim_buf_set_extmark()

api-fast deferred schedule Most API functions are deferred: they are queued ("scheduled") on the main loop and processed sequentially with normal input. If the editor is waiting for user input in a "modal" fashion (e.g. an input() prompt), a deferred request will block.

- Non-deferred (fast) functions such nvim_get_mode(), nvim_input(), or any Lua callback, are executed immediately (not sequenced in the input queue).
- Lua code can use vim.in_fast_event() to detect a fast context, where it may interact with Lua state but not "editor" state (textlock, options, window layout, ...).
- To perform editor operations, Lua code must schedule via vim.defer_fn() or vim.schedule(), or wait until `vim.in_fast_event()` returns `false`.

API metadata

api-metadata

The Nvim C API is automatically exposed to RPC by the build system, which parses headers in src/nvim/api/* and generates dispatch-functions mapping RPC API method names to public C API functions, converting/validating arguments and return values.

Nvim exposes its API metadata as a Dictionary with these items:

- version Nvim version, API level/compatibility
- version.api_level API version integer api-level
- version.api_compatible API is backwards-compatible with this level
- version.api_prerelease Declares the API as unstable/unreleased (`version.api_prerelease && fn.since == version.api_level`)
- functions API function signatures, containing api-types info describing the return value and parameters.
- ui_events UI event signatures
- ui_options Supported ui-options
- {fn}.since API level where function {fn} was introduced
- {fn}.deprecated_since API level where function {fn} was deprecated
- types Custom handle types defined by Nvim
- error_types Possible error types returned by API functions

About the `functions` map:

- Container types may be decorated with type/size constraints, e.g. ArrayOf(Buffer) or ArrayOf(Integer, 2).
- Each function `parameters` item is a `[type, name, optional]` tuple.

- Functions considered to be methods that operate on instances of Nvim special types (msgpack EXT) have the "method=true" flag. The receiver type is that of the first argument. Method names are prefixed with `nvim_` plus a type name, e.g. `nvim_buf_get_lines` is the `get_lines` method of a Buffer instance. dev-api
- Global functions have the "method=false" flag and are prefixed with just `nvim_`, e.g. `nvim_list_bufs`.

api-mapping

External programs (clients) can use the metadata to discover the API, using any of these approaches:

1. Connect to a running Nvim instance and call nvim_get_api_info() via msgpack-RPC. This is best for clients written in dynamic languages which can define functions at runtime.
2. Use the --api-info startup arg. Useful for statically-compiled clients. Example (requires Python "pyyaml" and "msgpack-python" modules):

```
nvim --api-info | python -c 'import msgpack, sys, yaml; yaml.dump(msgpack.unpackb(sys.stdin.buffer.read()), sys.stdout)'
```

3. Use the api_info() function.

```
:lua vim.print(vim.fn.api_info())
" Example using filter() to exclude non-deprecated API functions:
:new|put =map(filter(api_info().functions, '!has_key(v:val,''deprecated_since'')'), 'v:val.name')
```

API contract

api-contract

The Nvim API is composed of functions and events.

- Clients call functions like those described at api-global.
- Clients can subscribe to ui-events, api-buffer-updates, etc.
- API function names are prefixed with "nvim_".
- API event names are prefixed with "nvim_" and suffixed with "_event".

As Nvim evolves the API may change in compliance with this CONTRACT:

- New functions and events may be added.
  - Any such extensions are OPTIONAL: old clients may ignore them.
- New functions MAY CHANGE before release. Clients can dynamically check `api_prerelease`, api-metadata.
- Function signatures will NOT CHANGE after release, except as follows:
  - Map/list parameters/results may be EXTENDED (new fields may be added).
    - Such new fields are OPTIONAL: old clients MAY ignore them.
    - Existing fields will not be removed.
  - Return type MAY CHANGE from void to non-void. Old clients MAY ignore the new return value.
  - An optional `opts` parameter may be ADDED.
  - Optional parameters may be ADDED following an `opts` parameter.
  - The `opts` parameter and any following parameters MAY BE OMITTED by the client (equivalent to passing an empty dict). Explicitly passing a nil value is an error.
- Event parameters will not be removed or reordered (after release).
- Events may be EXTENDED: new parameters may be added.
- **Deprecated** functions will not be removed until Nvim 2.0.

"Private" interfaces are NOT covered by this contract:

- Undocumented (not in :help) functions or events of any kind
- nvim__x ("double underscore") functions

The idea is "versionless evolution", in the words of Rich Hickey:
- Relaxing a requirement should be a compatible change.
- Strengthening a promise should be a compatible change.

Buffer update events

api-buffer-updates

API clients can "attach" to Nvim buffers to subscribe to buffer update events. This is similar to TextChanged but more powerful and granular.

Call nvim_buf_attach() to receive these events on the channel:

nvim_buf_lines_event

nvim_buf_lines_event[`{buf}`, `{changedtick}`, `{firstline}`, `{lastline}`, `{linedata}`, `{more}`] When the buffer text between `{firstline}` and `{lastline}` (end-exclusive, zero-indexed) were changed to the new text in the `{linedata}` list. The granularity is a line, i.e. if a single character is changed in the editor, the entire line is sent.

When `{changedtick}` is v:null this means the screen lines (display) changed but not the buffer contents. `{linedata}` contains the changed screen lines. This happens when 'inccommand' shows a buffer preview.

Parameters:
- `{buf}` (`integer`) Buffer id
- `{changedtick}` (`integer`) Value of b:changedtick. If you send an API command back to Nvim you can check b:changedtick as part of your request to ensure that no other changes have been made.
- `{firstline}` (`integer`) The first line that was replaced. Zero-indexed: if line 1 was replaced then `{firstline}` will be zero, not one. Always less than or equal to the number of lines that were in the buffer before the lines were replaced.
- `{lastline}` (`integer`) The first line that was not replaced (i.e. the range `{firstline}`, `{lastline}` is end-exclusive). Zero-indexed: if line numbers 2 to 5 were replaced, this will be 5 instead of 6. Always less than or equal to the number of lines that were in the buffer before the lines were replaced. Will be -1 if the event is part of the initial update after attaching.
- `{linedata}` (`string[]`) Contents of the new buffer lines. Newline characters are omitted; empty lines are sent as empty strings.
- `{more}` (`boolean`) true for a "multipart" change notification: the current change was chunked into multiple nvim_buf_lines_event notifications (e.g. because it was too big).

nvim_buf_changedtick_event[{buf}, {changedtick}] nvim_buf_changedtick_event When b:changedtick was incremented but no text was changed. Relevant for undo/redo.

Parameters:
- {buf} (integer) Buffer id
- {changedtick} (integer) New value of b:changedtick.

nvim_buf_detach_event[{buf}] nvim_buf_detach_event
When buffer is detached (i.e. updates are disabled). Triggered explicitly by nvim_buf_detach() or implicitly in these cases:
- Buffer was abandoned and 'hidden' is not set.
- Buffer was reloaded, e.g. with :edit or an external change triggered :checktime or 'autoread'.
- Generally: whenever the buffer contents are unloaded from memory.

Parameters:
- {buf} (integer) Buffer id

EXAMPLE

Calling nvim_buf_attach() with send_buffer=true on an empty buffer, emits:

```
nvim_buf_lines_event[{buf}, {changedtick}, 0, -1, [""], v:false]
```

User adds two lines to the buffer, emits:

```
nvim_buf_lines_event[{buf}, {changedtick}, 0, 0, ["line1", "line2"], v:false]
```

User moves to a line containing the text "Hello world" and inserts "!", emits:

```
nvim_buf_lines_event[{buf}, {changedtick}, {linenr}, {linenr} + 1,
                     ["Hello world!"], v:false]
```

User moves to line 3 and deletes 20 lines using "20dd", emits:

```
nvim_buf_lines_event[{buf}, {changedtick}, 2, 22, [], v:false]
```

User selects lines 3-5 using linewise-visual mode and then types "p" to paste a block of 6 lines, emits:

```
nvim_buf_lines_event[{buf}, {changedtick}, 2, 5,
  ['pasted line 1', 'pasted line 2', 'pasted line 3', 'pasted line 4',
   'pasted line 5', 'pasted line 6'],
  v:false
]
```

User reloads the buffer with ":edit", emits:

```
nvim_buf_detach_event[{buf}]
```

LUA

api-buffer-updates-lua

In-process plugins can receive buffer updates via Lua callbacks. These callbacks are called frequently in various contexts; textlock prevents changing buffer contents and window layout (such operations must be scheduled). Moving the cursor is allowed, but it is restored afterwards.

nvim_buf_attach() will take keyword args for the callbacks. "on_lines" will receive parameters ("lines", {buf}, {changedtick}, {firstline}, {lastline}, {new_lastline}, {old_byte_size} [, {old_utf32_size}, {old_utf16_size}]). Unlike remote channel events the text contents are not passed. The new text can be accessed inside the callback as

```
vim.api.nvim_buf_get_lines(buf, firstline, new_lastline, true)
```

{old_byte_size} is the total size of the replaced region {firstline} to {lastline} in bytes, including the final newline after {lastline}. if utf_sizes is set to true in nvim_buf_attach() keyword args, then the UTF-32 and UTF-16 sizes of the deleted region is also passed as additional arguments {old_utf32_size} and {old_utf16_size}.

"on_changedtick" is invoked when b:changedtick was incremented but no text was changed. The parameters received are ("changedtick", {buf}, {changedtick}).

api-lua-detach

In-process Lua callbacks can detach by returning true. This will detach all callbacks attached with the same nvim_buf_attach() call without invoking the "on_detach" callback.

Buffer highlighting api-highlights

Nvim allows plugins to add position-based highlights to buffers. This is similar to matchaddpos() but with some key differences. The added highlights are associated with a buffer and adapts to line insertions and deletions, similar to signs. It is also possible to manage a set of highlights as a group and delete or replace all at once.

The intended use case are linter or semantic highlighter plugins that monitor a buffer for changes, and in the background compute highlights to the buffer. Another use case are plugins that show output in an append-only buffer, and want to add highlights to the outputs. Highlight data cannot be preserved on writing and loading a buffer to file, nor in undo/redo cycles.

Highlights are registered using the nvim_buf_set_extmark() function, which adds highlights as extmarks. If highlights need to be tracked or manipulated after adding them, the returned extmark id can be used.

```
-- create the highlight through an extmark
extid = vim.api.nvim_buf_set_extmark(buf, ns_id, line, col_start, {end_col = col_end, hl_group = hl_group})

-- example: modify the extmark's highlight group
vim.api.nvim_buf_set_extmark(buf, ns_id, line, col_start, {end_col = col_end, hl_group = NEW_HL_GROUP, id = extid})

-- example: change the highlight's position
vim.api.nvim_buf_set_extmark(buf, ns_id, NEW_LINE, col_start, {end_col = col_end, hl_group = NEW_HL_GROUP, id = extid})
```

See also vim.hl.range().

Floating windows

api-floatwin floating-windows E5601 E5602

Floating windows ("floats") are displayed on top of normal windows. This is useful to implement simple widgets, such as tooltips displayed next to the cursor. Floats are fully functional windows supporting user editing, common api-window calls, and most window options (except 'statusline').

Two ways to create a floating window:
- nvim_open_win() creates a new window (needs a buffer, see nvim_create_buf())
- nvim_win_set_config() reconfigures a normal window into a float

To close it use nvim_win_close() or a command such as :close.

To check whether a window is floating, check whether the `relative` option in its config is non-empty:

```
if vim.api.nvim_win_get_config(window_id).relative ~= '' then
  -- window with this window_id is floating
end
```

Buffer text can be highlighted by typical mechanisms (syntax highlighting, api-highlights). The hl-NormalFloat group highlights normal text; 'winhighlight' can be used as usual to override groups locally. Floats inherit options from the current window; specify `style=minimal` in nvim_open_win() to disable various visual features such as the 'number' column.

Other highlight groups specific to floating windows:
- hl-FloatBorder for window's border
- hl-FloatTitle for window's title
- hl-FloatFooter for window's footer

Currently, floating windows don't support some widgets like scrollbar.

The output of :mksession does not include commands for restoring floating windows.

Example: create a float with scratch buffer:

```
let buf = nvim_create_buf(v:false, v:true)
call nvim_buf_set_lines(buf, 0, -1, v:true, ["test", "text"])
let opts = {'relative': 'cursor', 'width': 10, 'height': 2, 'col': 0,
    \ 'row': 1, 'anchor': 'NW', 'style': 'minimal'}
let win = nvim_open_win(buf, 0, opts)
" optional: change highlight, otherwise Pmenu is used
call nvim_set_option_value('winhl', 'Normal:MyHighlight', {'win': win})
```

Extended marks

api-extended-marks extmarks extmark

Extended marks (extmarks) represent buffer annotations that track text changes in the buffer. They can represent cursors, folds, misspelled words, anything that needs to track a logical location in the buffer over time. api-indexing

Extmark position works like a "vertical bar" cursor: it exists between characters. Thus, the maximum extmark index on a line is 1 more than the character index:

```
 f o o b a r      line contents
 0 1 2 3 4 5      character positions (0-based)
0 1 2 3 4 5 6     extmark positions (0-based)
```

Extmarks have "forward gravity": if you place the cursor directly on an extmark position and enter some text, the extmark migrates forward.

```
f o o|b a r      line (| = cursor)
     3           extmark

f o o z|b a r    line (| = cursor)
       4         extmark (after typing "z")
```

If an extmark is on the last index of a line and you input a newline at that point, the extmark will accordingly migrate to the next line:

```
f o o z b a r|   line (| = cursor)
             7   extmark

f o o z b a r    first line
                 extmarks (none present)
|                second line (| = cursor)
0                extmark (after typing <CR>)
```

Example:

Let's set an extmark at the first row (row=0) and third column (column=2). api-indexing Passing id=0 creates a new mark and returns the id:

```
  01 2345678
0 ex|ample..
    ^ extmark position
```

```
let g:mark_ns = nvim_create_namespace('myplugin')
let g:mark_id = nvim_buf_set_extmark(0, g:mark_ns, 0, 2)
```

We can get the mark by its id:

```
echo nvim_buf_get_extmark_by_id(0, g:mark_ns, g:mark_id)
" => [0, 2]
```

We can get all marks in a buffer by namespace (or by a range):

```
echo nvim_buf_get_extmarks(0, g:mark_ns, 0, -1)
" => [[1, 0, 2]]
```

Deleting all surrounding text does NOT remove an extmark! To remove extmarks use nvim_buf_del_extmark(). Deleting "x" in our example:

```
  0 12345678
0 e|ample..
   ^ extmark position
```

```
echo nvim_buf_get_extmark_by_id(0, g:mark_ns, g:mark_id)
" => [0, 1]
```

Note: Extmark "gravity" decides how it will shift after a text edit. See nvim_buf_set_extmark()

Namespaces allow any plugin to manage only its own extmarks, ignoring those created by another plugin.

Extmark positions changed by an edit will be restored on undo/redo. Creating and deleting extmarks is not a buffer change, thus new undo states are not created for extmark changes.

Global Events

api-events

nvim_error_event({type}, {msg}) nvim_error_event
Emitted on the client channel if an async API request responds with an error.

Attributes:
RPC only

Parameters:
- {type} (integer) Error type id as defined by api_info().error_types.
- {msg} (string) Error message.

nvim_ui_term_event({event}, {value}) nvim_ui_term_event
Emitted by the TUI client to signal when a host-terminal event occurred.

Supports these events:
- "termresponse": The host-terminal sent an OSC, DCS, APC, or recognized CSI (DA1, kitty multiple-cursors) response to Nvim. The payload is the received response. Sets v:termresponse and fires TermResponse.

Attributes:
RPC only Since: 0.10.0

Parameters:
- {event} (string) Event name
- {value} (any) Event payload

Global Functions

api-global

nvim_chan_send({chan}, {data}) nvim_chan_send()
Sends raw data to channel chan. channel-bytes
- For a job, it writes it to the stdin of the process.
- For the stdio channel channel-stdio, it writes to Nvim's stdout.
- For an internal terminal instance (nvim_open_term()) it writes directly to terminal output.

This function writes raw data, not RPC messages. Use vim.rpcrequest() and vim.rpcnotify() if the channel expects RPC messages (i.e. it was created with rpc=true).

To write data to the TUI host terminal, see nvim_ui_send().

Attributes:
RPC only Lua vim.api only Since: 0.5.0

Parameters:
- {chan} (integer) Channel id
- {data} (string) Data to write. 8-bit clean: may contain NUL bytes.

nvim_chan_set({chan}, {config}) nvim_chan_set()
Sets channel properties.

Attributes:
api-fast Since: 0.13.0

Parameters:

- {chan} (integer) Channel id, or 0 for current (RPC) channel.
- {config} (vim.api.keyset.chan_set) Properties:
  - detach: (boolean, default: true) Closing the channel does not self-exit Nvim. Nvim self-exits when the last detach=false RPC channel closes. See also :detach!.

nvim_create_buf({listed}, {scratch}) nvim_create_buf()
Creates a new, empty, unnamed buffer.

Attributes:
Since: 0.4.0

Parameters:
- {listed} (boolean) Sets 'buflisted'
- {scratch} (boolean) Creates a "throwaway" scratch-buffer for temporary work (always 'nomodified'). Also sets 'nomodeline' on the buffer.

Return:
(integer) Buffer id, or 0 on error

nvim_del_current_line() nvim_del_current_line()
Deletes the current line.

Attributes:
not allowed when textlock is active Since: 0.1.0

nvim_del_keymap({mode}, {lhs}, {opts}) nvim_del_keymap()
Unmaps a global mapping for the given mode.

To unmap a buffer-local mapping, use nvim_buf_del_keymap().

Attributes:
Since: 0.4.0

Parameters:
- {mode} (string) Mode short-name ("n", "i", "v", ...)
- {lhs} (string) Left-hand-side {lhs} of the mapping.
- {opts} (vim.api.keyset.keymap_del?) Optional parameters.
  - lhs: When true, only match {lhs}, not {rhs}.

See also:
- nvim_set_keymap()

nvim_del_mark({name}) nvim_del_mark()
Deletes an uppercase/file named mark. See mark-motions.

Note:
- Lowercase name (or other buffer-local mark) is an error.

Attributes:
Since: 0.6.0

Parameters:
- {name} (string) Mark name

Return:
(boolean) true if the mark was deleted, else false.

See also:
- nvim_buf_del_mark()
- nvim_get_mark()

nvim_del_var({name}) nvim_del_var()
Removes a global (g:) variable.

Attributes:
Since: 0.1.0

Parameters:
- {name} (string) Variable name

nvim_echo({chunks}, {history}, {opts}) nvim_echo()
Prints a message given by a list of [text, hl_group] "chunks". Emits a Progress event if kind='progress'.

Returns a message-id, which can be given in later calls to update an existing message. The message-id is an autogenerated integer, or a user-defined string. The id "address space" is global, so plugins specifying a string id should use qualified names such as "my.msg.id" to avoid unintentional conflicts.

Example:

```
vim.api.nvim_echo({ { 'chunk1-line1\nchunk1-line2\n' }, { 'chunk2-line1' } }, true)
```

Attributes:
Since: 0.5.0

Parameters:
- {chunks} ([string, integer|string?][]) List of [text, hl_group] pairs, where each is a text string highlighted by the (optional) name or ID hl_group.
- {history} (boolean) if true, add to message-history.
- {opts} (vim.api.keyset.echo_opts?) Optional parameters.
  - data (table?) Dict of arbitrary data, available in Progress event-data.
  - err (boolean?) Treat the message like :echoerr. Sets hl_group to hl-ErrorMsg by default.

- id (`integer|string?`) Message-id returned by a previous `nvim_echo` call, or a user-defined id (string). If existing message has this id, it will be updated instead of creating a new message.
- kind (`string?`) Decides the ui-messages kind in the emitted message. Set "progress" to emit a progress-message.
- percent (`integer?`) progress-message percentage, or nil to signal "unknown progress".
- source (`string?`) progress-message source.
- status (`string?`) progress-message status:
  - "success": Process completed successfully.
  - "running": Process is ongoing.
  - "failed": Process failed.
  - "cancel": Process should be cancelled. Progress owner must handle the Progress event to perform the cancellation.
- title (`string?`) Message title. Only for progress-message currently.
- verbose (`boolean?`) Message is controlled by the 'verbose' option. `nvim -V3log` will write the message to the "log" file instead of standard output.

Return:
(`integer|string`) Message-id, or -1 if message wasn't shown.

nvim_eval_statusline({`str`}, {`opts`}) nvim_eval_statusline()
Evaluates statusline string.

Attributes:
api-fast Since: 0.6.0

Parameters:
- {`str`} (`string`) Statusline string (see 'statusline').
- {`opts`} (`vim.api.keyset.eval_statusline?`) Optional parameters.
  - fillchar: (string) Character to fill blank spaces in the statusline (see 'fillchars'). Treated as single-width even if it isn't.
  - highlights: (boolean) Return highlight information.
  - maxwidth: (number) Maximum width of statusline.
  - use_statuscol_lnum: (number) Evaluate statuscolumn for this line number instead of statusline.
  - use_tabline: (boolean) Evaluate tabline instead of statusline. When true, {`winid`} is ignored. Mutually exclusive with {`use_winbar`}.
  - use_winbar: (boolean) Evaluate winbar instead of statusline.
  - winid: (number) window-ID of the window to use as context for statusline.

Return:
(`vim.api.keyset.eval_statusline_ret`) Dict containing statusline information, with these keys:
- str: (string) Characters that will be displayed on the statusline.
- width: (number) Display width of the statusline.
- highlights: Array containing highlight information of the statusline. Only included when the "highlights" key in {`opts`} is true. Each element of the array is a Dict with these keys:
  - start: (number) Byte index (0-based) of first character that uses the highlight.
  - group: (string) **Deprecated**. Use `groups` instead.
  - groups: (array) Names of stacked highlight groups (highest priority last).

nvim_exec_lua({`code`}, {`args`}) nvim_exec_lua()
Executes Lua code. Arguments are available as `...` inside the chunk. The chunk can return a value.

Only statements are executed. To evaluate an expression, prefix it with "return": `return my_function(...)`

Example:

```
local peer = vim.fn.jobstart({ vim.v.progpath, '--clean', '--embed' }, { rpc=true })
vim.print(vim.rpcrequest(peer, 'nvim_exec_lua', [[
      local a, b = ...
      return ('result: %s'):format(a + b)
    ]],
    { 1, 3 }
  )
)
```

Attributes:
RPC only Since: 0.5.0

Parameters:
- {`code`} (`string`) Lua code to execute.
- {`args`} (`any[]`) Arguments to the Lua code.

Return:
(any) Value returned by the Lua code (if any), or NIL.

nvim_feedkeys({`keys`}, {`mode`}, {`escape_ks`}) nvim_feedkeys()
Sends input-keys to Nvim, subject to various quirks controlled by `mode` flags. This is a blocking call, unlike nvim_input().

On execution error: does not fail, but updates v:errmsg.

To input keycodes like `<C-o>`, pass the result of nvim_replace_termcodes():

```
:let key = nvim_replace_termcodes('<C-o>', v:true, v:false, v:true)
:call nvim_feedkeys(key, 'n', v:false)
```

Attributes:
Since: 0.1.0

Parameters:
- {`keys`} (`string`) Keys to send as input.
- {`mode`} (`string`) Behavior flags, see feedkeys().
- {`escape_ks`} (`boolean`) If true, escape K_SPECIAL bytes in `keys`. Should be false if you used nvim_replace_termcodes(), else true.

See also:
- feedkeys()
- vim_strsave_escape_ks

nvim_get_api_info() nvim_get_api_info()
Returns a 2-tuple (Array), where item 0 is the current channel id and item 1 is the api-metadata map (Dict).

Attributes:
api-fast RPC only Since: 0.1.0

Return:
([any, any]) 2-tuple [{channel-id}, {api-metadata}]

nvim_get_chan_info({chan}) nvim_get_chan_info()
Gets information about a channel.

The docs for nvim_list_uis() show an example.

Attributes:
Since: 0.3.0

Parameters:
- {chan} (integer) Channel id, or 0 for current (RPC) channel

Return:
(table<string,any>) Channel info dict with these keys:
- "argv" (optional) Job arguments list.
- "buf" (optional) Buffer connected to terminal instance.
- "buffer" (optional) **Deprecated** alias for buf.
- "client" (optional) Info about the peer (client on the other end of the channel), as set by nvim_set_client_info().
- "detach" (optional) Closing the RPC channel does not exit Nvim. nvim_chan_set()
- "exitcode" (optional) Exit code of the terminal process.
- "id" Channel id.
- "internal" (optional) In-process channel (stream=socket).
- "mode" How data received on the channel is interpreted.
  - "bytes" Send and receive raw bytes.
  - "terminal" terminal instance interprets ASCII sequences.
  - "rpc" RPC communication on the channel is active.
- "pty" (optional) Name of pseudoterminal. On a POSIX system this is a device path like "/dev/pts/1". If unknown, the key will still be present if a pty is used (e.g. for conpty on Windows).
- "stream" Stream underlying the channel.
  - "stdio" stdin and stdout of this Nvim instance
  - "stderr" stderr of this Nvim instance
  - "socket" TCP/IP socket or named pipe
  - "job" Job with communication over its stdio.

nvim_get_color_by_name({name}) nvim_get_color_by_name()
Returns the 24-bit RGB value of a nvim_get_color_map() color name or "#rrggbb" hexadecimal string.

Example:

```
:echo nvim_get_color_by_name("Pink")
:echo nvim_get_color_by_name("#cbcbcb")
```

Attributes:
Since: 0.1.0

Parameters:
- {name} (string) Color name or "#rrggbb" string

Return:
(integer) 24-bit RGB value, or -1 for invalid argument.

nvim_get_color_map() nvim_get_color_map()
Returns a map of color names and RGB values.

Keys are color names (e.g. "Aqua") and values are 24-bit RGB color values (e.g. 65535).

Attributes:
Since: 0.1.0

Return:
(table<string,integer>) Map of color names and RGB values.

nvim_get_context({opts}) nvim_get_context()
Gets a map (msgpack encoding) of the current editor state:
- "bufs" buffer-list
- "funcs" Vimscript global and script-local functions
- "gvars" global-variables
- "jumps" jumplist
- "regs" registers
- "sfuncs" Vimscript script-local functions

Attributes:
Since: 0.4.0

Parameters:
- {opts} (vim.api.keyset.context?) Optional parameters.

- types: List of context types (see above) to gather, or empty for "all".

Return:
(`table<string,any>`) Editor state as a msgpack-encoded map.

nvim_get_current_buf() nvim_get_current_buf()
Gets the current buffer.

Attributes:
Since: 0.1.0

Return:
(`integer`) Buffer id

nvim_get_current_line() nvim_get_current_line()
Gets the current line.

Attributes:
Since: 0.1.0

Return:
(`string`) Current line string

nvim_get_current_tabpage() nvim_get_current_tabpage()
Gets the current tabpage.

Attributes:
Since: 0.1.0

Return:
(`integer`) tab-ID

nvim_get_current_win() nvim_get_current_win()
Gets the current window.

Attributes:
Since: 0.1.0

Return:
(`integer`) window-ID

nvim_get_hl({`ns_id`}, {`opts`}) nvim_get_hl()
Gets all or specific highlight groups in a namespace.

Note:
- When the `link` attribute is defined in the highlight definition map, other attributes will not be taking effect (see :hi-link).

Attributes:
Since: 0.9.0

Parameters:
- {`ns_id`} (`integer`) Get highlight groups for namespace ns_id nvim_get_namespaces(). Use 0 to get global highlight groups :highlight.
- {`opts`} (`vim.api.keyset.get_highlight?`) Options dict:
  - create: (boolean, default true) When highlight group doesn't exist create it.
  - id: (integer) Get a highlight definition by id.
  - link: (boolean, default true) Show linked group name instead of effective definition :hi-link.
  - name: (string) Get a highlight definition by name.

Return:
(`vim.api.keyset.get_hl_info`) Highlight groups as a map from group name to a highlight definition map as in nvim_set_hl(), or only a single highlight definition map if requested by name or id.

nvim_get_hl_id_by_name({`name`}) nvim_get_hl_id_by_name()
Gets a highlight group by name

similar to hlID(), but allocates a new ID if not present.

Attributes:
Since: 0.5.0

Parameters:
- {`name`} (`string`)

Return:
(`integer`)

nvim_get_hl_ns({`opts`}) nvim_get_hl_ns()
Gets the active highlight namespace.

Attributes:
Since: 0.10.0

Parameters:
- {`opts`} (`vim.api.keyset.get_ns?`) Optional parameters
  - winid: (number) window-ID for retrieving a window's highlight namespace. A value of -1 is returned when nvim_win_set_hl_ns() has not been called for the window (or was called with a namespace of -1).

Return:
(`integer`) Namespace id, or -1

nvim_get_keymap({`mode`}) nvim_get_keymap()
Gets a list of global (non-buffer-local) mapping definitions.

Attributes:
Since: 0.2.1

Parameters:
- {mode} (string) Mode short-name ("n", "i", "v", ...)

Return:
(vim.api.keyset.get_keymap[]) Array of maparg()-like dictionaries describing mappings. The "buf" key is always zero.

nvim_get_mark({name}, {opts}) nvim_get_mark()
Returns a (row, col, buffer, buffername) tuple representing the position of the uppercase/file named mark. "End of line" column position is returned as v:maxcol (big number). See mark-motions.

Marks are (1,0)-indexed. api-indexing

Note:
- Lowercase name (or other buffer-local mark) is an error.

Attributes:
Since: 0.6.0

Parameters:
- {name} (string) Mark name
- {opts} (vim.api.keyset.empty?) Optional parameters. Reserved for future use.

Return:
([integer, integer, integer, string]) 4-tuple (row, col, buffer, buffername), (0, 0, 0, '') if the mark is not set.

See also:
- nvim_buf_set_mark()
- nvim_del_mark()

nvim_get_mode() nvim_get_mode()
Gets the current mode. mode() "blocking" is true if Nvim is waiting for input.

Attributes:
api-fast Since: 0.2.0

Return:
(vim.api.keyset.get_mode) Dict { "mode": String, "blocking": Boolean }

nvim_get_proc({pid}) nvim_get_proc()
Gets info describing process pid.

Attributes:
Since: 0.3.0

Parameters:
- {pid} (integer)

Return:
(any) Map of process properties, or NIL if process not found.

nvim_get_proc_children({pid}) nvim_get_proc_children()
Gets the immediate children of process pid.

Attributes:
Since: 0.3.0

Parameters:
- {pid} (integer)

Return:
(any[]) Array of child process ids, empty if process not found.

nvim_get_runtime_file({name}, {all}) nvim_get_runtime_file()
Finds files in runtime directories, in 'runtimepath' order.

"name" can contain wildcards. For example nvim_get_runtime_file("colors/*.{vim,lua}", true) will return all color scheme files. Always use forward slashes (/) in the search pattern for subdirectories regardless of platform.

It is not an error to not find any files. An empty array is returned then.

Attributes:
api-fast Since: 0.5.0

Parameters:
- {name} (string) pattern of files to search for
- {all} (boolean) whether to return all matches or only the first

Return:
(string[]) list of absolute paths to the found files

nvim_get_var({name}) nvim_get_var()
Gets a global (g:) variable.

Attributes:
Since: 0.1.0

Parameters:
- {name} (string) Variable name

Return:
(any) Variable value

nvim_get_vvar({name}) nvim_get_vvar()
Gets a v: variable.

Attributes:
Since: 0.1.0

Parameters:
- {name} (string) Variable name

Return:
(any) Variable value

nvim_input({keys}) nvim_input()
Queues raw user-input. Unlike nvim_feedkeys(), this uses a low-level input buffer and the call is non-blocking (input is processed asynchronously by the eventloop).

To input blocks of text, nvim_paste() is much faster and should be preferred.

On execution error: does not fail, but updates v:errmsg.

Note:
- keycodes like <CR> are translated, so "<" is special. To input a literal "<", send <LT>.
- For mouse events use nvim_input_mouse(). The pseudokey form <LeftMouse><col,row> is deprecated since api-level 6.

Attributes:
api-fast Since: 0.1.0

Parameters:
- {keys} (string) to be typed

Return:
(integer) Number of bytes actually written (can be fewer than requested if the buffer becomes full).

nvim_input_mouse() nvim_input_mouse()
nvim_input_mouse({button}, {action}, {modifier}, {grid}, {row}, {col}) Send mouse event from GUI.

Non-blocking: does not wait on any result, but queues the event to be processed soon by the event loop.

Note:
- Currently this doesn't support "scripting" multiple mouse events by calling it multiple times in a loop: the intermediate mouse positions will be ignored. It should be used to implement real-time mouse input in a GUI. The deprecated pseudokey form (<LeftMouse><col,row>) of nvim_input() has the same limitation.

Attributes:
api-fast Since: 0.4.0

Parameters:
- {button} (string) Mouse button: one of "left", "right", "middle", "wheel", "move", "x1", "x2".
- {action} (string) For ordinary buttons, one of "press", "drag", "release". For the wheel, one of "up", "down", "left", "right". Ignored for "move".
- {modifier} (string) String of modifiers each represented by a single char. The same specifiers are used as for a key press, except that the "-" separator is optional, so "C-A-", "c-a" and "CA" can all be used to specify Ctrl+Alt+click.
- {grid} (integer) Grid number (used by ui-multigrid client), or 0 to let Nvim decide positioning of windows. For more information, see dev-ui-multigrid
- {row} (integer) Mouse row-position (zero-based, like redraw events)
- {col} (integer) Mouse column-position (zero-based, like redraw events)

nvim_list_bufs() nvim_list_bufs()
Gets the current list of buffers.

Includes unlisted (unloaded/deleted) buffers, like :ls!. Use nvim_buf_is_loaded() to check if a buffer is loaded.

Attributes:
Since: 0.1.0

Return:
(integer[]) List of buffer ids

nvim_list_chans() nvim_list_chans()
Get information about all open channels.

Attributes:
Since: 0.3.0

Return:
(table<string,any>[]) Array of Dictionaries, each describing a channel with the format specified at nvim_get_chan_info().

nvim_list_runtime_paths() nvim_list_runtime_paths()
Gets the paths contained in runtime-search-path.

Attributes:
Since: 0.1.0

Return:
(string[]) List of paths

nvim_list_tabpages() nvim_list_tabpages()
Gets the current list of tab-IDs.

Attributes:
Since: 0.1.0

Return:

(`integer[]`) List of tab-IDs

nvim_list_uis() nvim_list_uis()
Gets a list of dictionaries representing attached UIs.

Example: The Nvim builtin TUI sets its channel info as described in startup-tui. In particular, it sets `client.name` to "nvim-tui". So you can check if the TUI is running by inspecting the client name of each UI:

```
vim.print(vim.api.nvim_get_chan_info(vim.api.nvim_list_uis()[1].chan).client.name)
```

Attributes:
Since: 0.3.0

Return:
(`table<string,any>[]`) Array of UI dictionaries, each with these keys:
- "height" Requested height of the UI
- "width" Requested width of the UI
- "rgb" true if the UI uses RGB colors (false implies cterm-colors)
- "ext_..." Requested UI extensions, see ui-option
- "chan" channel-id of remote UI

nvim_list_wins() nvim_list_wins()
Gets the current list of all window-IDs in all tabpages.

Attributes:
Since: 0.1.0

Return:
(`integer[]`) List of window-IDs

nvim_load_context({`dict`}) nvim_load_context()
Sets the current editor state from the given map.

Attributes:
Since: 0.4.0

Parameters:
- {`dict`} (`table<string,any>`) Msgpack-encoded editor state, in the form returned by nvim_get_context().

Return:
(any)

nvim_open_term({`buf`}, {`opts`}) nvim_open_term()
Open a terminal instance in a buffer

By default (and currently the only option) the terminal will not be connected to an external process. Instead, input sent on the channel will be echoed directly by the terminal. This is useful to display ANSI terminal sequences returned as part of an RPC message, or similar.

Note: :ls reports the buffer as "R" (running) until the channel is closed. chanclose()

Note: To initialize the terminal size, display the buffer in a window first. E.g. for a floating display,
1. Create an empty buffer using nvim_create_buf().
2. Display it with nvim_open_win().
3. Call nvim_open_term().
4. Then calling nvim_chan_send() will process sequences in a virtual terminal with the intended size (defined by the window width/height).

Example: this `TermHl` command can be used to display and highlight raw ANSI termcodes, so you can use Nvim as a "scrollback pager" (for terminals like kitty): ansi-colorize terminal-scrollback-pager

```
vim.api.nvim_create_user_command('TermHl', function()
  vim.api.nvim_open_term(0)
end, { desc = 'Highlights ANSI termcodes in curbuf' })
```

Attributes:
not allowed when textlock is active Since: 0.5.0

Parameters:
- {`buf`} (`integer`) Buffer which displays the PTY output. The initial buffer contents (if any) will be written to the PTY.
- {`opts`} (`vim.api.keyset.open_term?`) Optional parameters.
  - force_crlf: (boolean, default: true) Convert "\n" to "\r\n".
  - on_input: (`fun("input", chan: integer, buf: integer, data: string)`) Function invoked when the terminal emits bytes, see above. textlock applies. May call nvim_chan_send() directly.

Return:
(`integer`) Channel id, or 0 on error

nvim_paste({`data`}, {`crlf`}, {`phase`}) nvim_paste()
Pastes at cursor (in any mode), and sets "redo" so dot (.) will repeat the input. UIs call this to implement "paste", but it's also intended for use by scripts to input large, dot-repeatable blocks of text (as opposed to nvim_input() which is subject to mappings/events and is thus much slower).

Invokes the vim.paste() handler, which handles each mode appropriately.

Errors ('nomodifiable', `vim.paste()` failure, ...) are reflected in `err` but do not affect the return value (which is strictly decided by `vim.paste()`). On error or cancel, subsequent calls are ignored ("drained") until the next paste is initiated (phase 1 or -1).

Useful in mappings and scripts to insert multiline text. Example:

```
vim.keymap.set('n', 'x', function()
  vim.api.nvim_paste([[
    line1
    line2
    line3
  ]], false, -1)
end, { buf = 0 })
```

Attributes:
not allowed when textlock is active Since: 0.4.0

Parameters:
- {data} (string) Multiline input. Lines break at LF ("\n"). May be binary (containing NUL bytes).
- {crlf} (boolean) Also break lines at CR and CRLF.
- {phase} (integer) -1: paste in a single call (i.e. without streaming). To "stream" a paste, call nvim_paste sequentially with these phase values:
  - 1: starts the paste (exactly once)
  - 2: continues the paste (zero or more times)
  - 3: ends the paste (exactly once)

Return:
(boolean)
- true: Client may continue pasting.
- false: Client should cancel the paste.

nvim_put({lines}, {type}, {after}, {follow}) nvim_put()
Puts text at cursor, in any mode. For dot-repeatable input, use nvim_paste().

Compare :put and p which are always linewise.

Attributes:
not allowed when textlock is active Since: 0.4.0

Parameters:
- {lines} (string[]) readfile()-style list of lines. channel-lines
- {type} (string) Edit behavior: any getregtype() result, or:
  - "b" blockwise-visual mode (may include width, e.g. "b3")
  - "c" charwise mode
  - "l" linewise mode
  - "" guess by contents, see setreg()
- {after} (boolean) If true insert after cursor (like p), or before (like P).
- {follow} (boolean) If true place cursor at end of inserted text.

nvim_replace_termcodes()
nvim_replace_termcodes({str}, {from_part}, {do_lt}, {special}) Converts terminal codes and keycodes (<CR>, <Esc>, ...) in a key sequence, to the internal representation. See also Lua vim.keycode().

Attributes:
api-fast Since: 0.1.0

Parameters:
- {str} (string) String to be converted.
- {from_part} (boolean) Legacy Vim parameter. Usually true.
- {do_lt} (boolean) Also translate <lt>. Ignored if special is false.
- {special} (boolean) Replace keycodes, e.g. <CR> becomes a "\r" char.

Return:
(string)

See also:
- replace_termcodes
- cpoptions

nvim_select_popupmenu_item()
nvim_select_popupmenu_item({item}, {insert}, {finish}, {opts}) Selects an item in the completion popup menu.

If neither ins-completion nor cmdline-completion popup menu is active this API call is silently ignored. Useful for an external UI using ui-popupmenu to control the popup menu with the mouse. Can also be used in a mapping; use <Cmd> :map-cmd or a Lua mapping to ensure the mapping doesn't end completion mode.

Attributes:
Since: 0.4.0

Parameters:
- {item} (integer) Index (zero-based) of the item to select. Value of -1 selects nothing and restores the original text.
- {insert} (boolean) For ins-completion, whether the selection should be inserted in the buffer. Ignored for cmdline-completion.
- {finish} (boolean) Finish the completion and dismiss the popup menu. Implies {insert}.
- {opts} (vim.api.keyset.empty?) Optional parameters. Reserved for future use.

nvim_set_client_info()
nvim_set_client_info({name}, {version}, {type}, {methods}, {attributes}) Self-identifies the client, and sets optional flags on the channel. Defines the client object returned by nvim_get_chan_info().

Clients should call this just after connecting, to provide hints for debugging and orchestration. (**Note:** Something is better than nothing! Fields are optional, but at least set name.)

Can be called more than once; caller should merge old info if appropriate. Example: a library first identifies the channel, then a plugin using that library later identifies itself.

Attributes:
RPC only Since: 0.3.0

Parameters:
- {name} (string) Client short-name. Sets the client.name field of nvim_get_chan_info().
- {version} (table<string,any>) Dict describing the version, with these (optional) keys:
  - "commit" hash or similar identifier of commit
  - "major" major version (defaults to 0 if not set, for no release yet)
  - "minor" minor version
  - "patch" patch number
  - "prerelease" string describing a prerelease, like "dev" or "beta1"
- {type} (string) Must be one of the following values. Client libraries should default to "remote" unless overridden by the user.
  - "embedder" application using Nvim as a component (for example, IDE/editor implementing a vim mode).
  - "host" plugin host, typically started by nvim
  - "msgpack-rpc" remote client connected to Nvim via fully MessagePack-RPC compliant protocol.
  - "plugin" single plugin, started by nvim
  - "remote" remote client connected "Nvim flavored" MessagePack-RPC (responses must be in reverse order of requests). msgpack-rpc
  - "ui" gui frontend
- {methods} (table<string,any>) Builtin methods in the client. For a host, this does not include plugin methods which will be discovered later. The key should be the method name, the values are dicts with these (optional) keys (more keys may be added in future versions of Nvim, thus unknown keys are ignored. Clients must only use keys defined in this or later versions of Nvim):
  - "async" if true, send as a notification. If false or unspecified, use a blocking request
  - "nargs" Number of arguments. Could be a single integer or an array of two integers, minimum and maximum inclusive.
- {attributes} (table<string,any>) Arbitrary string:string map of informal client properties. Suggested keys:
  - "license": License description ("Apache 2", "GPLv3", "MIT", ...)
  - "logo": URI or path to image, preferably small logo or icon. .png or .svg format is preferred.
  - "pid": Process id.
  - "website": Client homepage URL (e.g. GitHub repository)

nvim_set_current_buf({buf}) nvim_set_current_buf()
Sets the current window's buffer to buf.

Attributes:
not allowed when textlock is active Since: 0.1.0

Parameters:
- {buf} (integer) Buffer id

nvim_set_current_dir({dir}) nvim_set_current_dir()
Changes the global working directory.

Attributes:
Since: 0.1.0

Parameters:
- {dir} (string) Directory path

nvim_set_current_line({line}) nvim_set_current_line()
Sets the text on the current line.

Attributes:
not allowed when textlock is active Since: 0.1.0

Parameters:
- {line} (string) Line contents

nvim_set_current_tabpage({tabpage}) nvim_set_current_tabpage()
Sets the current tabpage.

Attributes:
not allowed when textlock is active Since: 0.1.0

Parameters:
- {tabpage} (integer) tab-ID to focus

nvim_set_current_win({win}) nvim_set_current_win()
Navigates to the given window (and tabpage, implicitly).

Attributes:
not allowed when textlock is active Since: 0.1.0

Parameters:
- {win} (integer) window-ID to focus

nvim_set_hl({ns_id}, {name}, {val}) nvim_set_hl()
Sets a highlight group. By default, replaces the entire definition (e.g. nvim_set_hl(0, 'Visual', {}) will clear the "Visual" group), unless update is specified.

Note:
- The fg and bg keys also accept the string values "fg" or "bg" which act as aliases to the corresponding foreground and background values of the Normal group. If the Normal group has not been defined, using these values results in an error.
- If link is used in combination with other attributes; only the link will take effect (see :hi-link).

Attributes:
Since: 0.5.0

Parameters:

- {ns_id} (integer) Namespace id for this highlight nvim_create_namespace(). Use 0 to set a highlight group globally :highlight. Highlights from non-global namespaces are not active by default, use nvim_set_hl_ns() or nvim_win_set_hl_ns() to activate them.
- {name} (string) Highlight group name, e.g. "ErrorMsg"
- {val} (vim.api.keyset.highlight) Highlight definition map, accepts the following keys:
  - altfont: boolean
  - bg: color name or "#RRGGBB", see note.
  - bg_indexed: boolean. If true, bg is an RGB approximation of ctermbg (a palette index). UIs rendering cterm natively may prefer ctermbg.
  - blend: integer between 0 and 100
  - blink: boolean
  - bold: boolean
  - conceal: boolean Concealment at the UI level (terminal SGR), unrelated to :syn-conceal.
  - cterm: cterm attribute map, like highlight-args. If not set, cterm attributes will match those from the attribute map documented above.
  - ctermbg: Sets background of cterm color ctermbg
  - ctermfg: Sets foreground of cterm color ctermfg
  - default: boolean Don't override existing definition :hi-default
  - dim: boolean
  - fg: Color name or "#RRGGBB", see note.
  - fg_indexed: boolean. Same as bg_indexed, for fg and ctermfg.
  - font: GUI font name (string). Sets highlight-font. Use "NONE" to clear.
  - force: (boolean, default: false) Update the highlight group even if it already exists.
  - italic: boolean
  - link: Name of highlight group to link to. :hi-link
  - link_global: Like "link", but always resolved in the global namespace (ns=0).
  - nocombine: boolean
  - overline: boolean
  - reverse: boolean
  - sp: Color name or "#RRGGBB"
  - standout: boolean
  - strikethrough: boolean
  - undercurl: boolean
  - underdashed: boolean
  - underdotted: boolean
  - underdouble: boolean
  - underline: boolean
  - update: (boolean, default: false) Update specified attributes only, leave others unchanged.

nvim_set_hl_ns({ns_id}) nvim_set_hl_ns()
Set active namespace for highlights defined with nvim_set_hl(). This can be set for a single window, see nvim_win_set_hl_ns().

Attributes:
Since: 0.8.0

Parameters:
- {ns_id} (integer) the namespace to use

nvim_set_hl_ns_fast({ns_id}) nvim_set_hl_ns_fast()
Set active namespace for highlights defined with nvim_set_hl() while redrawing.

This function is meant to be called while redrawing, primarily from nvim_set_decoration_provider() on_win and on_line callbacks, which are allowed to change the namespace during a redraw cycle.

Attributes:
api-fast Since: 0.8.0

Parameters:
- {ns_id} (integer) the namespace to activate

nvim_set_keymap({mode}, {lhs}, {rhs}, {opts}) nvim_set_keymap()
Sets a global mapping for the given mode.

To set a buffer-local mapping, use nvim_buf_set_keymap().

Unlike :map, leading/trailing whitespace is accepted as part of the {lhs} or {rhs}. Empty {rhs} is <Nop>. keycodes are replaced as usual.

Example:

```
call nvim_set_keymap('n', ' <NL>', '', {'nowait': v:true})
```

is equivalent to:

```
nmap <nowait> <Space><NL> <Nop>
```

Attributes:
Since: 0.4.0

Parameters:
- {mode} (string) Mode short-name (map command prefix: "n", "i", "v", "x", …) or "!" for :map!, or empty string for :map. "ia", "ca" or "!a" for abbreviation in Insert mode, Cmdline mode, or both, respectively
- {lhs} (string) Left-hand-side {lhs} of the mapping.
- {rhs} (string) Right-hand-side {rhs} of the mapping.
- {opts} (vim.api.keyset.keymap?) Optional parameters map: Accepts all :map-arguments as keys except <buffer>, values are booleans (default false). Also:
  - "callback" Lua function called in place of {rhs}.

- "desc" human-readable description.
- "noremap" disables recursive_mapping, like :noremap
- "replace_keycodes" (boolean) When "expr" is true, replace keycodes in the resulting string (see nvim_replace_termcodes()). Returning nil from the Lua "callback" is equivalent to returning an empty string.

nvim_set_var({name}, {value}) nvim_set_var()
Sets a global (g:) variable.

Attributes:
Since: 0.1.0

Parameters:
- {name} (string) Variable name
- {value} (any) Variable value

nvim_set_vvar({name}, {value}) nvim_set_vvar()
Sets a v: variable, if it is not readonly.

Attributes:
Since: 0.4.0

Parameters:
- {name} (string) Variable name
- {value} (any) Variable value

nvim_strwidth({text}) nvim_strwidth()
Calculates the number of display cells occupied by text. Control characters including <Tab> count as one cell.

Attributes:
Since: 0.1.0

Parameters:
- {text} (string) Some text

Return:
(integer) Number of cells

nvim__cmdwin_set({type}, {buf}) nvim__cmdwin_set()
WARNING: This feature is experimental/unstable.

Records the cmdwin scratchbuf and type, or clears both when type="" / buf=0. Internal use only.

Attributes:
Since: 0.12.0

Parameters:
- {type} (string) ':', '/', '?' (first char only); empty to clear.
- {buf} (integer) cmdwin buffer id, or 0 to clear.

nvim__complete_set({index}, {opts}) nvim__complete_set()
WARNING: This feature is experimental/unstable.

Sets info for the completion item at the given index. If the info text was shown in a window, returns the window and buffer ids, or empty dict if not shown.

Parameters:
- {index} (integer) Completion candidate index
- {opts} (vim.api.keyset.complete_set?) Optional parameters.
  - info: (string) info text.

Return:
(table<string,integer>) Dict containing these keys:
- winid: (integer) floating window id
- bufnr: (integer) buffer id in floating window

nvim__exec_lua_fast({code}, {args}) nvim__exec_lua_fast()
WARNING: This feature is experimental/unstable.

Like nvim_exec_lua(), but can be called during api-fast contexts.

Execute Lua code. Parameters (if any) are available as ... inside the chunk. The chunk can return a value.

Only statements are executed. To evaluate an expression, prefix it with return: return my_function(...)

Attributes:
api-fast RPC only Since: 0.12.0

Parameters:
- {code} (string) Lua code to execute
- {args} (any[]) Arguments to the code

Return:
(any) Return value of Lua code if present or NIL.

nvim__get_runtime({pat}, {all}, {opts}) nvim__get_runtime()
WARNING: This feature is experimental/unstable.

Find files in runtime directories

Attributes:
api-fast Since: 0.6.0

Parameters:

- {pat} (string[]) pattern of files to search for
- {all} (boolean) whether to return all matches or only the first
- {opts} (vim.api.keyset.runtime?) is_lua: only search Lua subdirs

Return:
(string[]) list of absolute paths to the found files

nvim__id({obj}) nvim__id()
WARNING: This feature is experimental/unstable.

Returns object given as argument.

This API function is used for testing. One should not rely on its presence in plugins.

Parameters:
- {obj} (any) Object to return.

Return:
(any) its argument.

nvim__id_array({arr}) nvim__id_array()
WARNING: This feature is experimental/unstable.

Returns array given as argument.

This API function is used for testing. One should not rely on its presence in plugins.

Parameters:
- {arr} (any[]) Array to return.

Return:
(any[]) its argument.

nvim__id_dict({dct}) nvim__id_dict()
WARNING: This feature is experimental/unstable.

Returns dict given as argument.

This API function is used for testing. One should not rely on its presence in plugins.

Parameters:
- {dct} (table<string,any>) Dict to return.

Return:
(table<string,any>) its argument.

nvim__id_float({flt}) nvim__id_float()
WARNING: This feature is experimental/unstable.

Returns floating-point value given as argument.

This API function is used for testing. One should not rely on its presence in plugins.

Parameters:
- {flt} (number) Value to return.

Return:
(number) its argument.

nvim__inspect_cell({grid}, {row}, {col}) nvim__inspect_cell()
WARNING: This feature is experimental/unstable.

NB: if your UI doesn't use hlstate, this will not return hlstate first time.

Parameters:
- {grid} (integer)
- {row} (integer)
- {col} (integer)

Return:
(any[])

nvim__invalidate_glyph_cache() nvim__invalidate_glyph_cache()
WARNING: This feature is experimental/unstable.

For testing. The condition in schar_cache_clear_if_full is hard to reach, so this function can be used to force a cache clear in a test.

nvim__redraw({opts}) nvim__redraw()
WARNING: This feature is experimental/unstable.

Instruct Nvim to redraw various components.

Attributes:
Since: 0.10.0

Parameters:
- {opts} (vim.api.keyset.redraw?) Optional parameters.
  - buf: Target a specific buffer number as described below.
  - cursor: Immediately update cursor position on the screen in win or the current window.
  - flush: Update the screen with pending updates.
  - range: Redraw a range in buf, the buffer in win or the current buffer (useful for decoration providers). Expects a tuple [first, last] with the first and last line number of the range, 0-based end-exclusive api-indexing.
  - statuscolumn: Redraw the 'statuscolumn' in buf, win or all windows.
  - statusline: Redraw the 'statusline' in buf, win or all windows.

- tabline: Redraw the 'tabline'.
- valid: When present mark `win`, `buf`, or all windows for redraw. When `true`, only redraw changed lines (useful for decoration providers). When `false`, forcefully redraw.
- win: Target a specific window-ID as described below.
- winbar: Redraw the 'winbar' in `buf`, `win` or all windows.

See also:
- :redraw

nvim__set_restart_on_crash() nvim__set_restart_on_crash({`progpath`}, {`argv`}) **WARNING:** This feature is experimental/unstable.

Parameters:
- {`progpath`} (`string`)
- {`argv`} (`any[]`)

nvim__stats() nvim__stats()
WARNING: This feature is experimental/unstable.

Gets internal stats.

Return:
(`table<string,any>`) Map of various internal stats.

Vimscript Functions api-vimscript

nvim_call_dict_function() nvim_call_dict_function({`dict`}, {`fn`}, {`args`}) Calls a Vimscript Dictionary-function with the given arguments.

On execution error: fails with Vimscript error, updates v:errmsg.

Attributes:
Since: 0.3.0

Parameters:
- {`dict`} (any) Dict, or String evaluating to a Vimscript self dict
- {`fn`} (`string`) Name of the function defined on the Vimscript dict
- {`args`} (`any[]`) Function arguments packed in an Array

Return:
(any) Result of the function call

nvim_call_function({`fn`}, {`args`}) nvim_call_function()
Calls a Vimscript function with the given arguments.

On execution error: fails with Vimscript error, updates v:errmsg.

Attributes:
Since: 0.1.0

Parameters:
- {`fn`} (`string`) Function to call
- {`args`} (`any[]`) Function arguments packed in an Array

Return:
(any) Result of the function call

nvim_command({`cmd`}) nvim_command()
Executes an Ex command.

On execution error: fails with Vimscript error, updates v:errmsg.

Prefer nvim_cmd() or nvim_exec2() instead. To modify an Ex command in a structured way before executing it, modify the result of nvim_parse_cmd() then pass it to nvim_cmd().

Attributes:
Since: 0.1.0

Parameters:
- {`cmd`} (`string`) Ex command string

nvim_eval({`expr`}) nvim_eval()
Evaluates a Vimscript expression. Dicts and Lists are recursively expanded.

On execution error: fails with Vimscript error, updates v:errmsg.

Attributes:
Since: 0.1.0

Parameters:
- {`expr`} (`string`) Vimscript expression string

Return:
(any) Evaluation result or expanded object

nvim_exec2({`src`}, {`opts`}) nvim_exec2()
Executes Vimscript (multiline block of Ex commands), like anonymous :source.

Unlike nvim_command() this function supports heredocs, script-scope (s:), etc.

On execution error: fails with Vimscript error, updates v:errmsg.

Attributes:

Since: 0.9.0

Parameters:
- {src} (string) Vimscript code
- {opts} (vim.api.keyset.exec_opts?) Optional parameters.
  - output: (boolean, default false) Whether to capture and return all (non-error, non-shell :!) output.

Return:
(table<string,any>) Dict containing information about execution, with these keys:
- output: (string|nil) Output if opts.output is true.

See also:
- execute()
- nvim_command()
- nvim_cmd()

nvim_parse_expression({expr}, {flags}, {hl}) nvim_parse_expression()
Parse a Vimscript expression.

Attributes:
api-fast Since: 0.3.0

Parameters:
- {expr} (string) Expression to parse. Always treated as a single line.
- {flags} (string) Flags:
  - "E" if EOC tokens are not allowed (determines whether they will stop parsing process or be recognized as an operator/space, though also yielding an error).
  - "l" when needing to start parsing with lvalues for ":let" or ":for".
  - "m" if multiple expressions in a row are allowed (only the first one will be parsed),
  - Common flag sets:
    - "E" to parse like for "<C-r>=".
    - "lm" to parse for ":let".
    - "m" to parse like for ":echo".
    - empty string for ":call".
- {hl} (boolean) If true, return value will also include "highlight" key containing array of 4-tuples (arrays) (Integer, Integer, Integer, String), where first three numbers define the highlighted region and represent line, starting column and ending column (latter exclusive: one should highlight region [start_col, end_col)).

Return:
(table<string,any>)
- AST: top-level dict with these keys:
  - "error": Dict with error, present only if parser saw some error. Contains the following keys:
    - "message": String, error message in printf format, translated. Must contain exactly one %.*s.
    - "arg": String, error message argument.
  - "len": Amount of bytes successfully parsed. With flags equal to "" that should be equal to the length of expr string. ("Successfully parsed" here means "participated in AST creation", not "till the first error".)
  - "ast": AST, either nil or a dict with these keys:
    - "type": node type, one of the value names from ExprASTNodeType stringified without "kExprNode" prefix.
    - "start": a pair [line, column] describing where node is "started" where "line" is always 0 (will not be 0 if you will be using this API on e.g. ":let", but that is not present yet). Both elements are Integers.
    - "len": “length” of the node. This and "start" are there for debugging purposes primary (debugging parser and providing debug information).
    - "children": a list of nodes described in top/"ast". There always is zero, one or two children, key will not be present if node has no children. Maximum number of children may be found in node_maxchildren array.
- Local values (present only for certain nodes):
  - "scope": a single Integer, specifies scope for "Option" and "PlainIdentifier" nodes. For "Option" it is one of ExprOptScope values, for "PlainIdentifier" it is one of ExprVarScope values.
  - "ident": identifier (without scope, if any), present for "Option", "PlainIdentifier", "PlainKey" and "Environment" nodes.
  - "name": Integer, register name (one character) or -1. Only present for "Register" nodes.
  - "cmp_type": String, comparison type, one of the value names from ExprComparisonType, stringified without "kExprCmp" prefix. Only present for "Comparison" nodes.
  - "ccs_strategy": String, case comparison strategy, one of the value names from ExprCaseCompareStrategy, stringified without "kCCStrategy" prefix. Only present for "Comparison" nodes.
  - "augmentation": String, augmentation type for "Assignment" nodes. Is either an empty string, "Add", "Subtract" or "Concat" for "=", "+=", "-=" or ".=" respectively.
  - "invert": Boolean, true if result of comparison needs to be inverted. Only present for "Comparison" nodes.
  - "ivalue": Integer, integer value for "Integer" nodes.
  - "fvalue": Float, floating-point value for "Float" nodes.
  - "svalue": String, value for "SingleQuotedString" and "DoubleQuotedString" nodes.

Autocmd Functions api-autocmd

nvim_clear_autocmds({opts}) nvim_clear_autocmds()
Clears all autocommands matching the {opts} query. To delete autocmds see nvim_del_autocmd().

Attributes:
Since: 0.7.0

Parameters:
- {opts} (vim.api.keyset.clear_autocmds?) Optional parameters:
  - buf: (integer?) Select autocmd-buflocal autocommands. Not allowed with {pattern}.
  - event: (vim.api.keyset.events|vim.api.keyset.events[]?) Examples:

    - event: "pat1"
    - event: { "pat1" }
    - event: { "pat1", "pat2", "pat3" }
  - group: (`string|int?`) Group name or id.
    - **NOTE:** If not given, matches autocmds not in any group.
  - pattern: (`string|table?`) Filter by patterns (exact match). Not allowed with `{buf}`.
    - Example: if you have `*.py` as that pattern for the autocmd, you must pass `*.py` exactly to clear it. `test.py` will not match the pattern.

nvim_create_augroup({name}, {opts}) nvim_create_augroup()
Create or get an autocommand group autocmd-groups.

To get an existing group id, do:

```
local id = vim.api.nvim_create_augroup('my.lsp.config', {
  clear = false
})
```

Attributes:
Since: 0.7.0

Parameters:
- `{name}` (`string`) Group name
- `{opts}` (`vim.api.keyset.create_augroup?`) Optional parameters:
  - clear (`boolean?`, default: true) Clear existing commands in the group autocmd-groups.

Return:
(`integer`) Group id.

See also:
- autocmd-groups

nvim_create_autocmd({event}, {opts}) nvim_create_autocmd()
Creates an autocommand event handler, defined by `callback` (Lua function or Vimscript function name string) or `command` (Ex command string).

Example using Lua callback:

```
vim.api.nvim_create_autocmd({'BufEnter', 'BufWinEnter'}, {
  pattern = {'*.c', '*.h'},
  callback = function(ev)
    print(string.format('event fired: %s', vim.inspect(ev)))
  end
})
```

Example using an Ex command as the handler:

```
vim.api.nvim_create_autocmd({'BufEnter', 'BufWinEnter'}, {
  pattern = {'*.c', '*.h'},
  command = "echo 'Entering a C or C++ file'",
})
```

Note: `pattern` is NOT automatically expanded (unlike with :autocmd), thus names like "$HOME" and "~" must be expanded explicitly:

```
pattern = vim.fn.expand('~') .. '/some/path/*.py'
```

Attributes:
api-fast Since: 0.7.0

Parameters:
- `{event}` (`vim.api.keyset.events|vim.api.keyset.events[]`) Event(s) that will trigger the handler (`callback` or `command`): one handler is created for each event name.
- `{opts}` (`vim.api.keyset.create_autocmd?`) Options dict:
  - buf (`integer?`) Buffer id for buffer-local autocommands autocmd-buflocal. Not allowed with `{pattern}`.
  - callback (`function|string?`) Lua function (or Vimscript function name, if string) called when the event(s) is triggered. Lua callback can return lua-truthy to delete the autocommand. Callback receives one argument, a table with keys: event-args
    - buf: (`integer`) <abuf>
    - data: (any) Arbitrary data passed from nvim_exec_autocmds() event-data
    - event: (`vim.api.keyset.events`) Name of the triggered event autocmd-events
    - file: (`string`) <afile> (not expanded to a full path)
    - group: (`integer?`) Group id, if any
    - id: (`integer`) Autocommand id
    - match: (`string`) <amatch> (expanded to a full path)
    - win: (`integer`) id of the window for the event, see window-ID
  - command (string?) Vim command executed on event. Not allowed with `{callback}`.
  - desc (`string?`) Description (for documentation and troubleshooting).
  - group (`string|integer?`) Group name or id to match against.
  - nested (`boolean?`, default: false) Run nested autocommands autocmd-nested.
  - once (`boolean?`, default: false) Handle the event only once autocmd-once. If `{event}` is a list, each handler will fire once.
  - pattern (`string|array?`) Pattern(s) to match literally autocmd-pattern.

Return:
(`integer`) Autocommand id (number)

See also:
- autocommand

- nvim_del_autocmd()

nvim_del_augroup_by_id({id}) nvim_del_augroup_by_id()
Delete an autocommand group by id.

To get a group id one can use nvim_get_autocmds().

NOTE: behavior differs from :augroup-delete. When deleting a group, autocommands contained in this group will also be deleted and cleared. This group will no longer exist.

Attributes:
Since: 0.7.0

Parameters:
- {id} (integer) Group id.

See also:
- nvim_del_augroup_by_name()
- nvim_create_augroup()

nvim_del_augroup_by_name({name}) nvim_del_augroup_by_name()
Delete an autocommand group by name.

NOTE: behavior differs from :augroup-delete. When deleting a group, autocommands contained in this group will also be deleted and cleared. This group will no longer exist.

Attributes:
Since: 0.7.0

Parameters:
- {name} (string) Group name.

See also:
- autocmd-groups

nvim_del_autocmd({id}) nvim_del_autocmd()
Deletes an autocommand by id.

Attributes:
Since: 0.7.0

Parameters:
- {id} (integer) Autocommand id returned by nvim_create_autocmd()

nvim_exec_autocmds({event}, {opts}) nvim_exec_autocmds()
Executes {event} handlers matching the {opts} query, in the context of {buf} (if given). autocmd-execute

Attributes:
Since: 0.7.0

Parameters:
- {event} (vim.api.keyset.events|vim.api.keyset.events[]) Event(s) to execute.
- {opts} (vim.api.keyset.exec_autocmds?) Optional filters:
  - buf (integer?) Buffer where the event is applied. autocmd-buflocal Not allowed with {pattern}.
  - data (any): Arbitrary data passed to the callback. See nvim_create_autocmd().
  - group (string|integer?) Group name or id to match against. autocmd-groups.
  - modeline (boolean?, default: true) Process the modeline after the autocommands <nomodeline>. Ignored if buf is given.
  - pattern (string|array?, default: current file name) autocmd-pattern. Not allowed with {buf}.

See also:
- :doautocmd

nvim_get_autocmds({opts}) nvim_get_autocmds()
Gets all autocommands matching ALL criteria in the {opts} query.

Note: When multiple patterns or events are provided they have "OR" semantics (any combination is matched).

Examples:

```
-- Matches all criteria
autocommands = vim.api.nvim_get_autocmds({
  group = 'MyGroup',
  event = {'BufEnter', 'BufWinEnter'},
  pattern = {'*.c', '*.h'}
})

-- All commands from one group
autocommands = vim.api.nvim_get_autocmds({
  group = 'MyGroup',
})
```

Attributes:
Since: 0.7.0

Parameters:
- {opts} (vim.api.keyset.get_autocmds?) Dict with at least one of these keys:
  - buf: (integer[]|integer?) Buffer id or list of buffer ids, for buffer-local autocommands autocmd-buflocal. Not allowed with {pattern}.
  - event: (vim.api.keyset.events|vim.api.keyset.events[]?) Event(s) to match autocmd-events.
  - group: (string|table?) Group name or id to match.
  - id: (integer?) Autocommand ID to match.
  - pattern: (string|table?) Pattern(s) to match autocmd-pattern. Not allowed with {buf}.

Return:
(`vim.api.keyset.get_autocmds.ret[]`) Array of matching autocommands, where each item has:
- buf (`integer?`): Buffer id (only for autocmd-buffer-local).
- buflocal (`boolean?`): true if the autocommand is buffer-local autocmd-buffer-local.
- callback: (`function|string?`): Event handler: a Lua function or Vimscript function name.
- command: (`string`) Event handler: an Ex-command. Empty if a `callback` is set.
- desc: (`string`) Description.
- event: (`vim.api.keyset.events`) Event name(s).
- group: (`integer`) Group id.
- group_name: (`string`) Group name.
- id: (`integer`) Autocommand id (only when defined with the API).
- once: (`boolean`) true if autocmd-once was set.
- pattern: (`string`) Autocommand pattern.

Buffer Functions

api-buffer

For more information on buffers, see buffers.

Unloaded Buffers:

Buffers may be unloaded by the :bunload command or the buffer's 'bufhidden' option. When a buffer is unloaded its file contents are freed from memory and vim cannot operate on the buffer lines until it is reloaded (usually by opening the buffer again in a new window). API methods such as nvim_buf_get_lines() and nvim_buf_line_count() will be affected.

You can use nvim_buf_is_loaded() or nvim_buf_line_count() to check whether a buffer is loaded.

nvim_buf_attach({`buf`}, {`send_buffer`}, {`opts`}) nvim_buf_attach()
Activates api-buffer-updates events on a channel, or as Lua callbacks.

Example (Lua): capture buffer updates in a global `events` variable (use "vim.print(events)" to see its contents):

```
events = {}
vim.api.nvim_buf_attach(0, false, {
  on_lines = function(...)
    table.insert(events, {...})
  end,
})
```

Attributes:
Since: 0.3.0

Parameters:
- {`buf`} (`integer`) Buffer id, or 0 for current buffer
- {`send_buffer`} (`boolean`) True if the initial notification should contain the whole buffer: first notification will be `nvim_buf_lines_event`. Else the first notification will be `nvim_buf_changedtick_event`. Not for Lua callbacks.
- {`opts`} (`vim.api.keyset.buf_attach?`) Optional parameters.
  - on_bytes: Called on granular changes (compared to on_lines). Not called on buffer reload (`:checktime`, `:edit`, ...), see `on_reload`. Returning lua-truthy deletes the callback. Args:
    - the string "bytes"
    - buffer id
    - b:changedtick
    - start row of the changed text (zero-indexed)
    - start column of the changed text
    - byte offset of the changed text (from the start of the buffer)
    - old end row of the changed text (offset from start row)
    - old end column of the changed text (if old end row = 0, offset from start column)
    - old end byte length of the changed text
    - new end row of the changed text (offset from start row)
    - new end column of the changed text (if new end row = 0, offset from start column)
    - new end byte length of the changed text
  - on_changedtick: Called on changetick increment without text change. Args:
    - the string "changedtick"
    - buffer id
    - b:changedtick
  - on_detach: Called on detach, or when the buffer is unloaded or deleted. Not called when a callback returns lua-truthy to delete itself. Args:
    - the string "detach"
    - buffer id
  - on_lines: Called on linewise changes. Not called on buffer reload (`:checktime`, `:edit`, ...), see `on_reload`. Returning lua-truthy deletes the callback. Args:
    - the string "lines"
    - buffer id
    - b:changedtick
    - first line that changed (zero-indexed)
    - last line that was changed
    - last line in the updated range
    - byte count of previous contents
    - deleted_codepoints (if `utf_sizes` is true)
    - deleted_codeunits (if `utf_sizes` is true)
  - on_reload: Called on whole-buffer load (`:checktime`, `:edit`, ...). Clients should typically re-fetch the entire buffer contents. Args:
    - the string "reload"

    - buffer id
  - preview: also attach to command preview (i.e. 'inccommand') events.
  - utf_sizes: include UTF-32 and UTF-16 size of the replaced region, as args to `on_lines`.

Return:
(`boolean`) False if attach failed (invalid parameter, or buffer isn't loaded); otherwise True.

See also:
- nvim_buf_detach()
- api-buffer-updates-lua

nvim_buf_call({`buf`}, {`fn`}) nvim_buf_call()
Calls function `fn` in the context of buffer `buf` and returns its result (may be multiple values).

This temporarily switches current buffer to `buf`. If the current window already shows `buf`, the window is not switched. If a window inside the current tabpage (including a float) already shows the buffer, then one of those windows will be set as current window temporarily. Otherwise a temporary scratch window (called the "autocmd window" for historical reasons) will be used.

This is useful e.g. to call Vimscript functions that only work with the current buffer/window currently, like `jobstart(…, {'term': v:true})`.

Attributes:
Lua vim.api only Since: 0.5.0

Parameters:
- {`buf`} (`integer`) Buffer id, or 0 for current buffer.
- {`fn`} (`function`) Lua function to call inside the buffer.

Return:
(`any`) Value(s) returned by `fn()`.

nvim_buf_del_keymap()
nvim_buf_del_keymap({`buf`}, {`mode`}, {`lhs`}, {`opts`}) Unmaps a buffer-local mapping for the given mode.

Attributes:
Since: 0.4.0

Parameters:
- {`buf`} (`integer`) Buffer id, or 0 for current buffer
- {`mode`} (`string`) Mode short-name ("n", "i", "v", ...)
- {`lhs`} (`string`) Left-hand-side {lhs} of the mapping.
- {`opts`} (`vim.api.keyset.keymap_del?`) Optional parameters.
  - lhs: When true, only match {`lhs`}, not {`rhs`}.

See also:
- nvim_del_keymap()

nvim_buf_del_mark({`buf`}, {`name`}) nvim_buf_del_mark()
Deletes a named mark in the buffer. See mark-motions.

Note:
- only deletes marks set in the buffer, if the mark is not set in the buffer it will return false.

Attributes:
Since: 0.6.0

Parameters:
- {`buf`} (`integer`) Buffer to set the mark on
- {`name`} (`string`) Mark name

Return:
(`boolean`) true if the mark was deleted, else false.

See also:
- nvim_buf_set_mark()
- nvim_del_mark()

nvim_buf_del_var({`buf`}, {`name`}) nvim_buf_del_var()
Removes a buffer-scoped (b:) variable

Attributes:
Since: 0.1.0

Parameters:
- {`buf`} (`integer`) Buffer id, or 0 for current buffer
- {`name`} (`string`) Variable name

nvim_buf_delete({`buf`}, {`opts`}) nvim_buf_delete()
Deletes a buffer and its metadata (like :bwipeout).

To get :bdelete behavior, reset 'buflisted' and pass `unload=true`:

```
vim.bo.buflisted = false
vim.api.nvim_buf_delete(0, { unload = true })
```

Attributes:
not allowed when textlock is active Since: 0.5.0

Parameters:
- {`buf`} (`integer`) Buffer id, or 0 for current buffer
- {`opts`} (`vim.api.keyset.buf_delete?`) Optional parameters. Keys:
  - force: Force deletion, ignore unsaved changes.

- unload: Unloaded only (:bunload), do not delete.

nvim_buf_detach({buf}) nvim_buf_detach()
Deactivates buffer-update events on the channel.

Attributes:
RPC only Since: 0.3.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer

Return:
(boolean) False if detach failed (because the buffer isn't loaded); otherwise True.

See also:
- nvim_buf_attach()
- api-lua-detach for detaching Lua callbacks

nvim_buf_get_changedtick({buf}) nvim_buf_get_changedtick()
Gets a changed tick of a buffer

Attributes:
Since: 0.2.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer

Return:
(integer) b:changedtick value.

nvim_buf_get_keymap({buf}, {mode}) nvim_buf_get_keymap()
Gets a list of buffer-local mapping definitions.

Attributes:
Since: 0.2.1

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {mode} (string) Mode short-name ("n", "i", "v", ...)

Return:
(vim.api.keyset.get_keymap[]) Array of maparg()-like dictionaries describing mappings. The "buf" key holds the associated buffer id.

nvim_buf_get_lines() nvim_buf_get_lines({buf}, {start}, {end}, {strict_indexing}) Gets a line-range from the buffer.

Indexing is zero-based, end-exclusive. Negative indices are interpreted as length+1+index: -1 refers to the index past the end. So to get the last element use start=-2 and end=-1.

Out-of-bounds indices are clamped to the nearest valid value, unless strict_indexing is set.

Attributes:
Since: 0.1.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {start} (integer) First line index
- {end} (integer) Last line index, exclusive
- {strict_indexing} (boolean) Whether out-of-bounds should be an error.

Return:
(string[]) Array of lines, or empty array for unloaded buffer.

See also:
- nvim_buf_get_text()

nvim_buf_get_mark({buf}, {name}) nvim_buf_get_mark()
Returns a (row,col) tuple representing the position of the named mark. "End of line" column position is returned as v:maxcol (big number). See mark-motions.

Marks are (1,0)-indexed. api-indexing

Attributes:
Since: 0.1.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {name} (string) Mark name

Return:
([integer, integer]) (row, col) tuple, (0, 0) if the mark is not set, or is an uppercase/file mark set in another buffer.

See also:
- nvim_buf_set_mark()
- nvim_buf_del_mark()

nvim_buf_get_name({buf}) nvim_buf_get_name()
Gets the full/absolute filepath of the buffer, or the buffer name for non-file buffers.

If the buffer represents a directory, the name ends with a path separator, unless it was changed by :file or nvim_buf_set_name().

Attributes:
Since: 0.1.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer

Return:
(string) Buffer name

nvim_buf_get_offset({buf}, {index}) nvim_buf_get_offset()
Returns the byte offset of a line (0-indexed). api-indexing

Line 1 (index=0) has offset 0. UTF-8 bytes are counted. EOL is one byte. 'fileformat' and 'fileencoding' are ignored. The line index just after the last line gives the total byte-count of the buffer. A final EOL byte is counted if it would be written, see 'eol'.

Unlike line2byte(), throws error for out-of-bounds indexing. Returns -1 for unloaded buffer.

Attributes:
Since: 0.3.2

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {index} (integer) Line index

Return:
(integer) Integer byte offset, or -1 for unloaded buffer.

nvim_buf_get_text()
nvim_buf_get_text({buf}, {start_row}, {start_col}, {end_row}, {end_col}, {opts}) Gets a range from the buffer (may be partial lines, unlike nvim_buf_get_lines()).

Indexing is zero-based. Row indices are end-inclusive, and column indices are end-exclusive.

Prefer nvim_buf_get_lines() when retrieving entire lines.

Attributes:
Since: 0.7.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {start_row} (integer) First line index
- {start_col} (integer) Starting column (byte offset) on first line
- {end_row} (integer) Last line index, inclusive
- {end_col} (integer) Ending column (byte offset) on last line, exclusive
- {opts} (vim.api.keyset.empty?) Optional parameters. Currently unused.

Return:
(string[]) Array of lines, or empty array for unloaded buffer.

nvim_buf_get_var({buf}, {name}) nvim_buf_get_var()
Gets a buffer-scoped (b:) variable.

Attributes:
Since: 0.1.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {name} (string) Variable name

Return:
(any) Variable value

nvim_buf_is_loaded({buf}) nvim_buf_is_loaded()
Checks if a buffer is valid and loaded. See api-buffer for more info about unloaded buffers.

Attributes:
Since: 0.3.2

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer

Return:
(boolean) true if the buffer is valid and loaded, false otherwise.

nvim_buf_is_valid({buf}) nvim_buf_is_valid()
Checks if a buffer is valid.

Note:
- Even if a buffer is valid it may have been unloaded. See api-buffer for more info about unloaded buffers.

Attributes:
Since: 0.1.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer

Return:
(boolean) true if the buffer is valid, false otherwise.

nvim_buf_line_count({buf}) nvim_buf_line_count()
Returns the number of lines in the given buffer.

Attributes:
Since: 0.1.0

Parameters:

- {buf} (integer) Buffer id, or 0 for current buffer

Return:
(integer) Line count, or 0 for unloaded buffer. api-buffer

nvim_buf_set_keymap() nvim_buf_set_keymap({buf}, {mode}, {lhs}, {rhs}, {opts}) Sets a buffer-local mapping for the given mode.

Attributes:
Since: 0.4.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {mode} (string)
- {lhs} (string)
- {rhs} (string)
- {opts} (vim.api.keyset.keymap?)

See also:
- nvim_set_keymap()

nvim_buf_set_lines() nvim_buf_set_lines({buf}, {start}, {end}, {strict_indexing}, {replacement}) Sets (replaces) a line-range in the buffer.

Indexing is zero-based, end-exclusive. Negative indices are interpreted as length+1+index: -1 refers to the index past the end. So to change or delete the last line use start=-2 and end=-1.

To insert lines at a given index, set start and end to the same index. To delete a range of lines, set replacement to an empty array.

Out-of-bounds indices are clamped to the nearest valid value, unless strict_indexing is set.

Attributes:
not allowed when textlock is active Since: 0.1.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {start} (integer) First line index
- {end} (integer) Last line index, exclusive
- {strict_indexing} (boolean) Whether out-of-bounds should be an error.
- {replacement} (string[]) Array of lines to use as replacement

See also:
- nvim_buf_set_text()

nvim_buf_set_mark() nvim_buf_set_mark({buf}, {name}, {line}, {col}, {opts}) Sets a named mark in the given buffer, all marks are allowed file/uppercase, visual, last change, etc. See mark-motions.

Marks are (1,0)-indexed. api-indexing

Note:
- Passing 0 as line deletes the mark

Attributes:
Since: 0.6.0

Parameters:
- {buf} (integer) Buffer to set the mark on
- {name} (string) Mark name
- {line} (integer) Line number
- {col} (integer) Column/row number
- {opts} (vim.api.keyset.empty?) Optional parameters. Reserved for future use.

Return:
(boolean) true if the mark was set, else false.

See also:
- nvim_buf_del_mark()
- nvim_buf_get_mark()

nvim_buf_set_name({buf}, {name}) nvim_buf_set_name()
Sets the full file name for a buffer, like :file_f

Attributes:
Since: 0.1.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {name} (string) Buffer name

nvim_buf_set_text() nvim_buf_set_text({buf}, {start_row}, {start_col}, {end_row}, {end_col}, {replacement}) Sets (replaces) a range in the buffer

This is recommended over nvim_buf_set_lines() when only modifying parts of a line, as extmarks will be preserved on non-modified parts of the touched lines.

Indexing is zero-based. Row indices are end-inclusive, and column indices are end-exclusive.

To insert text at a given (row, column) location, use start_row = end_row = row and start_col = end_col = col. To delete the text in a range, use replacement = {}.

Note:

- Prefer nvim_buf_set_lines() (for performance) to add or delete entire lines.
- Prefer nvim_paste() or nvim_put() to insert (instead of replace) text at cursor.

Attributes:
not allowed when textlock is active Since: 0.5.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {start_row} (integer) First line index
- {start_col} (integer) Starting column (byte offset) on first line
- {end_row} (integer) Last line index, inclusive
- {end_col} (integer) Ending column (byte offset) on last line, exclusive
- {replacement} (string[]) Array of lines to use as replacement

nvim_buf_set_var({buf}, {name}, {value}) nvim_buf_set_var()
Sets a buffer-scoped (b:) variable

Attributes:
Since: 0.1.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {name} (string) Variable name
- {value} (any) Variable value

Command Functions api-command

nvim_buf_create_user_command()
nvim_buf_create_user_command({buf}, {name}, {cmd}, {opts}) Creates a buffer-local command user-commands.

Attributes:
Since: 0.7.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer.
- {name} (string)
- {cmd} (any)
- {opts} (vim.api.keyset.user_command?)

See also:
- nvim_create_user_command

nvim_buf_del_user_command({buf}, {name}) nvim_buf_del_user_command()
Delete a buffer-local user-defined command.

Only commands created with :command-buffer or nvim_buf_create_user_command() can be deleted with this function.

Attributes:
Since: 0.7.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer.
- {name} (string) Name of the command to delete.

nvim_buf_get_commands({buf}, {opts}) nvim_buf_get_commands()
Gets a map of buffer-local user-commands.

Attributes:
Since: 0.3.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {opts} (vim.api.keyset.get_commands?) Optional parameters. Currently not used.

Return:
(vim.api.keyset.command_info) Map of maps describing commands.

nvim_cmd({cmd}, {opts}) nvim_cmd()
Executes an Ex command cmd, specified as a Dict with the same structure as returned by nvim_parse_cmd().

Use magic={…=false} to disable special chars (see also open-file):

```
vim.api.nvim_cmd({
    cmd = 'edit',
    args = { '%foo"|bar#baz"' },
    magic = { file = false, bar = false }
  },
  {}
)
```

- See nvim_parse_cmd() to parse a cmdline string (which can then be passed to nvim_cmd).
- See nvim_command() to execute a cmdline string.

On execution error: fails with Vimscript error, updates v:errmsg.

Attributes:
Since: 0.8.0

Parameters:

- {cmd} (vim.api.keyset.cmd) Command to execute, a Dict with the same structure as the return value of nvim_parse_cmd() (except "addr", "nargs" and "nextcmd" are ignored). All keys except "cmd" are optional.
- {opts} (vim.api.keyset.cmd_opts?) Optional parameters.
  - output: (boolean, default false) Whether to return command output.

Return:
(string) Command output (non-error, non-shell :!) if output is true, else empty string.

See also:
- nvim_command()
- nvim_exec2()
- nvim_parse_cmd()

nvim_create_user_command() nvim_create_user_command({name}, {cmd}, {opts}) Creates a global user-commands command.

For Lua usage see lua-guide-commands-create.

Example:

```
:call nvim_create_user_command('SayHello', 'echo "Hello world!"', {'bang': v:true})
:SayHello
Hello world!
```

Attributes:
Since: 0.7.0

Parameters:
- {name} (string) Command name. First char must be uppercase.
- {cmd} (string|fun(args: vim.api.keyset.create_user_command.command_args)) Command or Lua function, executed when the command is invoked. Lua function receives a table with keys:
  - args: (string) Args passed to the command, if any. <args>
  - bang: (boolean) true if the command was executed with "!". <bang>
  - count: (number) Count, if any. <count>
  - fargs: (table) Args split by unescaped whitespace (when more than one arg is allowed), if any. <f-args>
  - line1: (number) Start of the command range. <line1>
  - line2: (number) End of the command range. <line2>
  - mods: (string) Command modifiers (unstructured string), if any. <mods>
  - name: (string) Command name.
  - nargs: (string) Number of arguments allowed by the command. :command-nargs
  - range: (number) Number of items in the command range: 0, 1, or 2. <range>
  - reg: (string) Register name, if any. <reg>
  - smods: (table) Command modifiers (structured), same as "mods" in nvim_parse_cmd().
- {opts} (vim.api.keyset.user_command?) Optional flags:
  - addr :command-addr
  - complete :command-complete command or function :command-completion-customlist.
  - count :command-count
  - desc (string) Command description.
  - force (boolean, default true) Override the existing definition, if any.
  - nargs Number of arguments allowed by the command. :command-nargs
  - preview (function) Preview handler for 'inccommand'. :command-preview
  - range :command-range
  - boolean command-attributes such as :command-bang or :command-bar (but not :command-buffer, use nvim_buf_create_user_command() instead).

nvim_del_user_command({name}) nvim_del_user_command()
Delete a user-defined command.

Attributes:
Since: 0.7.0

Parameters:
- {name} (string) Name of the command to delete.

nvim_get_commands({opts}) nvim_get_commands()
Gets a map of global (non-buffer-local) Ex commands.

Currently only user-commands are supported, not builtin Ex commands.

Attributes:
Since: 0.3.0

Parameters:
- {opts} (vim.api.keyset.get_commands?) Optional parameters. Currently only supports {"builtin":false}

Return:
(table<string,vim.api.keyset.command_info>) Map of maps describing commands.

See also:
- nvim_get_all_options_info()

nvim_parse_cmd({str}, {opts}) nvim_parse_cmd()
Parse command line.

Doesn't check the validity of command arguments.

Attributes:
api-fast Since: 0.8.0

Parameters:
- {str} (string) Command line string to parse. Cannot contain "\n".
- {opts} (vim.api.keyset.empty?) Optional parameters. Reserved for future use.

Return:
(vim.api.keyset.cmd) Dict containing command information, with these keys:
- cmd: (string) Command name.
- range: (array) (optional) Command range (<line1> <line2>). Omitted if command doesn't accept a range. Otherwise, has no elements if no range was specified, one element if only a single range item was specified, or two elements if both range items were specified.
- count: (number) (optional) Command <count>. Omitted if command cannot take a count.
- reg: (string) (optional) Command <register>. Omitted if command cannot take a register.
- bang: (boolean) Whether command contains a <bang> (!) modifier.
- args: (array) Command arguments.
- addr: (string) Value of :command-addr. Uses short name or "line" for -addr=lines.
- nargs: (string) Value of :command-nargs.
- nextcmd: (string) Next command if there are multiple commands separated by a :bar. Empty if there isn't a next command.
- magic: (dict) Which characters have special meaning in the command arguments.
  - file: (boolean) The command expands filenames. Which means characters such as "%", "#" and wildcards are expanded.
  - bar: (boolean) The "|" character is treated as a command separator and the double quote character (") is treated as the start of a comment.
- mods: (dict) :command-modifiers.
  - filter: (dict) :filter.
    - pattern: (string) Filter pattern. Empty string if there is no filter.
    - force: (boolean) Whether filter is inverted or not.
  - silent: (boolean) :silent.
  - emsg_silent: (boolean) :silent!.
  - unsilent: (boolean) :unsilent.
  - sandbox: (boolean) :sandbox.
  - noautocmd: (boolean) :noautocmd.
  - browse: (boolean) :browse.
  - confirm: (boolean) :confirm.
  - hide: (boolean) :hide.
  - horizontal: (boolean) :horizontal.
  - keepalt: (boolean) :keepalt.
  - keepjumps: (boolean) :keepjumps.
  - keepmarks: (boolean) :keepmarks.
  - keeppatterns: (boolean) :keeppatterns.
  - lockmarks: (boolean) :lockmarks.
  - noswapfile: (boolean) :noswapfile.
  - tab: (integer) :tab. -1 when omitted.
  - verbose: (integer) :verbose. -1 when omitted.
  - vertical: (boolean) :vertical.
  - split: (string) Split modifier string, is an empty string when there's no split modifier. If there is a split modifier it can be one of:
    - "aboveleft": :aboveleft.
    - "belowright": :belowright.
    - "topleft": :topleft.
    - "botright": :botright.

Extmark Functions **api-extmark**

nvim_buf_clear_namespace()
nvim_buf_clear_namespace({buf}, {ns_id}, {line_start}, {line_end}) Clears namespaced objects (highlights, extmarks, virtual text) from a region.

Lines are 0-indexed. api-indexing To clear the namespace in the entire buffer, specify line_start=0 and line_end=-1.

Attributes:
Since: 0.3.2

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {ns_id} (integer) Namespace to clear, or -1 to clear all namespaces.
- {line_start} (integer) Start of range of lines to clear
- {line_end} (integer) End of range of lines to clear (exclusive) or -1 to clear to end of buffer.

nvim_buf_del_extmark({buf}, {ns_id}, {id}) nvim_buf_del_extmark()
Removes an extmark.

Attributes:
Since: 0.5.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {ns_id} (integer) namespace id
- {id} (integer) Extmark id

Return:
(boolean) true if the extmark was found, else false

nvim_buf_get_extmark_by_id()
nvim_buf_get_extmark_by_id({buf}, {ns_id}, {id}, {opts}) Gets the position (0-indexed) of an extmark.

Attributes:
Since: 0.5.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {ns_id} (integer) namespace id
- {id} (integer) Extmark id
- {opts} (vim.api.keyset.get_extmark?) Optional parameters. Keys:
  - details: Whether to include the details dict
  - hl_name: Whether to include highlight group name instead of id, true if omitted

Return:
([integer, integer, vim.api.keyset.extmark_details?]) 0-indexed (row, col, details?) tuple or empty list () if extmark id was absent. The optional details dictionary contains the same keys as opts in nvim_buf_set_extmark(), except for id, conceal_lines and ephemeral. It also contains the following keys:
- ns_id: namespace id
- invalid: boolean that indicates whether the mark is hidden because the entirety of text span range is deleted. See also the key invalidate in nvim_buf_set_extmark().

nvim_buf_get_extmarks()

nvim_buf_get_extmarks({buf}, {ns_id}, {start}, {end}, {opts}) Gets extmarks in "traversal order" from a charwise region defined by buffer positions (inclusive, 0-indexed api-indexing).

Region can be given as (row,col) tuples, or valid extmark ids (whose positions define the bounds). 0 and -1 are understood as (0,0) and (-1,-1) respectively, thus the following are equivalent:

```
vim.api.nvim_buf_get_extmarks(0, my_ns, 0, -1)
vim.api.nvim_buf_get_extmarks(0, my_ns, {0,0}, {-1,-1})
```

If end is less than start, marks are returned in reverse order. (Useful with limit, to get the first marks prior to a given position.)

Note:
- For a reverse range, limit does not actually affect the traversed range, just how many marks are returned
- When using extmark ranges (marks with a end_row/end_col position) the overlap option might be useful. Otherwise only the start position of an extmark will be considered.
- The :marks command can list extmarks.
- Legacy signs placed through the :sign commands are implemented as extmarks. Their details array will contain a sign_name field.

Example:

```
local api = vim.api
local pos = api.nvim_win_get_cursor(0)
local ns  = api.nvim_create_namespace('my-plugin')
-- Create new extmark at line 1, column 1.
local m1  = api.nvim_buf_set_extmark(0, ns, 0, 0)
-- Create new extmark at line 3, column 1.
local m2  = api.nvim_buf_set_extmark(0, ns, 2, 0)
-- Get extmarks only from line 3.
local ms  = api.nvim_buf_get_extmarks(0, ns, {2,0}, {2,0})
-- Get all marks in this buffer + namespace.
local all = api.nvim_buf_get_extmarks(0, ns, 0, -1)
vim.print(ms)
```

Attributes:
Since: 0.5.0

Parameters:
- {buf} (integer) Buffer id, or 0 for current buffer
- {ns_id} (integer) namespace id, or -1 for all namespaces
- {start} (any) Start of range: a 0-indexed (row, col) or valid extmark id (whose position defines the bound). api-indexing
- {end} (any) End of range (inclusive): a 0-indexed (row, col) or valid extmark id (whose position defines the bound). api-indexing
- {opts} (vim.api.keyset.get_extmarks?) Optional parameters. Keys:
  - details: Whether to include the details dict
  - hl_name: Whether to include highlight group name instead of id, true if omitted
  - limit: Maximum number of marks to return
  - overlap: Also include marks which overlap the range, even if their start position is less than start
  - type: Filter marks by type: "highlight", "sign", "virt_text" and "virt_lines"

Return:
(vim.api.keyset.get_extmark_item[]) List of [extmark_id, row, col, details?] tuples in "traversal order". For the details dictionary, see nvim_buf_get_extmark_by_id().

nvim_buf_set_extmark()

nvim_buf_set_extmark({buf}, {ns_id}, {line}, {col}, {opts}) Creates or updates an extmark.

By default a new extmark is created when no id is passed in, but it is also possible to create a new mark by passing in a previously unused id or move an existing mark by passing in its id. The caller must then keep track of existing and unused ids itself. (Useful over RPC, to avoid waiting for the return value.)

Using the optional arguments, it is possible to use this to highlight a range of text, and also to associate virtual text to the mark.

If present, the position defined by end_col and end_row should be after the start position in order for the extmark to cover a range. An earlier end position is not an error, but then it behaves like an empty range (no highlighting).

Attributes:
Since: 0.5.0

Parameters:

- {buf} (integer) Buffer id, or 0 for current buffer
- {ns_id} (integer) namespace id
- {line} (integer) Line where to place the mark, 0-based. api-indexing
- {col} (integer) Column where to place the mark, 0-based. api-indexing
- {opts} (vim.api.keyset.set_extmark?) Optional parameters:
  - conceal: (boolean|string)
    - string: Single char or empty string, enables conceal similar to :syn-conceal. Non-empty char is used as :syn-cchar. Highlighted with "hl_group" if defined, else defaults to hl-Conceal.
    - boolean: true is equivalent to "", false removes any existing conceal.
  - conceal_lines: (string) Line-level conceal. When set to an empty string (other values reserved for future use), the lines in the extmark range are not drawn; the next non-concealed line is drawn in their place. Requires 'conceallevel' >= 2; subject to 'concealcursor' on the cursor line.
  - cursorline_hl_group: Highlight group used for the 'signcolumn' text when the cursor is on the same line as the mark and 'cursorline' is enabled.
  - end_col: End-column of the mark, 0-based exclusive, or -1 to extend the range to end of line (if strict=false).
  - end_right_gravity: (boolean, default: false) Indicates which direction the extmark end-position (if any) will be shifted when new text is inserted (true=right, false=left).
  - end_row: End-line of the mark, 0-based inclusive.
  - ephemeral: For use with nvim_set_decoration_provider() callbacks. The mark will only be used for the current redraw cycle, and not stored in the buffer.
  - hl_eol: When true, for a multiline highlight covering the EOL of a line, continue the highlight for the rest of the screen line (just like for diff and cursorline highlight).
  - hl_group: Highlight group for the text range. This and below highlight groups can be given as a string or as an integer (nvim_get_hl_id_by_name()). Multiple highlight groups can be stacked by passing an array (highest-priority last).
  - hl_mode: Controls how highlights are combined with the highlights of the text. Only affects virt_text highlights, but might affect `hl_group` in the future.
    - "replace": Only show the virt_text color. This is the default.
    - "combine": Combine with background text color.
    - "blend": Blend with background text color. Not supported for "inline" virt_text.
  - id: Update this extmark instead of creating a new one.
  - invalidate: (boolean) Indicates whether to hide the extmark if the entirety of its range is deleted. For hidden marks, an "invalid" key is added to the "details" array of nvim_buf_get_extmarks() and family. If "undo_restore" is false, the extmark is deleted instead.
  - line_hl_group: Highlight group used for the whole line.
  - number_hl_group: Highlight group used for the number column.
  - priority: Priority of the highlight group, sign attribute or virtual text. For virtual text, the highest-priority item is drawn last. For example treesitter highlighting uses a value of 100.
  - right_gravity: (boolean, default: true) Indicates which direction the extmark will be shifted when new text is inserted (true=right, false=left).
  - sign_hl_group: Highlight group for `sign_text`.
  - sign_text: (string, length 1-2) Displayed in 'signcolumn'.
  - spell: (boolean) Indicates that spell checking should be performed within this extmark.
  - strict: (boolean, default: true) Indicates extmark should not be placed if the line or column value is past end-of-buffer or end-of-line respectively.
  - ui_watched: (boolean) Indicates the mark should be drawn by a UI; activates `win_extmark` ui-multigrid events. **Note:** the mark is positioned by virt_text attributes. Can be used with virt_text.
  - undo_restore: (default: true) Restore the exact position of the mark if text around the mark was deleted and then restored by undo.
  - url: URL stored on this extmark. In the TUI, the OSC 8 control sequence is used to generate a clickable hyperlink to this URL.
  - virt_lines: List of virtual lines to place next to the mark, where each line is a list of `[text_chunk, hl]` tuples. Buffer and window options do not affect the display of the text, except 'tabstop' is used for hard tabs. The 'wrap' and 'linebreak' options do not take effect, so the number of extra screen lines will match the list-size (but see `virt_lines_overflow`). By default lines are placed below the marked line. Special case: If the last `text_chunk` is empty, the corresponding `hl` will extend to the end of the line.
  - virt_lines_above: Place virtual lines above instead.
  - virt_lines_leftcol: Place virtual lines in the leftmost column of the window, bypassing sign and number columns.
  - virt_lines_overflow: Controls display of virtual lines exceeding the viewport:
    - "auto": Decided by the 'wrap' option.
    - "scroll": Scroll horizontally with 'nowrap', otherwise the same as "trunc".
    - "trunc": Truncate on the right (default).
    - "wrap": Always wrap (regardless of the 'wrap' option).
  - virt_text: virtual-text stored and displayed by this mark. List of `[text, hl]` tuples, each representing a text chunk with a highlight `hl` which may be a single highlight group or a list of stacked highlight groups (highest-priority last).
  - virt_text_hide: Hide the virtual text when the background text is selected or hidden because of scrolling with 'nowrap' or 'smoothscroll'. Currently only affects "overlay" virt_text.
  - virt_text_pos: Position of virtual text:
    - "eol": One cell after the line text or its displayed 'listchars' EOL character (default).
    - "eol_right_align": Display right-aligned in the window unless the virtual text is longer than the space available. If the virtual text is too long, it is truncated to fit in the window after the EOL character. If the line is wrapped, the virtual text is shown after the end of the line rather than the previous screen line.
    - "overlay": Display over the specified column, without shifting the underlying text.
    - "right_align": Display right-aligned in the window.
    - "inline": Display at the specified column, and right-shift the buffer text as needed.
  - virt_text_repeat_linebreak: Repeat the virtual text on wrapped lines.
  - virt_text_win_col: Position the virtual text at a fixed window column (starting from the first text column of the screen line) instead of "virt_text_pos".

Return:

(integer) Id of the created/updated extmark

nvim_create_namespace({name}) nvim_create_namespace()
Creates a new namespace or gets an existing one. namespace

Namespaces are used for buffer highlights and virtual text, see nvim_buf_set_extmark().

Namespaces can be named or anonymous. If name matches an existing namespace, the associated id is returned. If name is an empty string a new, anonymous namespace is created.

Attributes:
Since: 0.3.2

Parameters:
- {name} (string) Namespace name or empty string

Return:
(integer) Namespace id

nvim_get_namespaces() nvim_get_namespaces()
Gets existing, non-anonymous namespaces.

Attributes:
Since: 0.3.2

Return:
(table<string,integer>) dict that maps from names to namespace ids.

nvim_set_decoration_provider()
nvim_set_decoration_provider({ns_id}, {opts}) Set or change decoration provider for a namespace

This is a very general purpose interface for having Lua callbacks being triggered during the redraw code.

The expected usage is to set extmarks for the currently redrawn buffer. nvim_buf_set_extmark() can be called to add marks on a per-window or per-lines basis. Use the ephemeral key to only use the mark for the current screen redraw (the callback will be called again for the next redraw).

Note: this function should not be called often. Rather, the callbacks themselves can be used to throttle unneeded callbacks. the on_start callback can return false to disable the provider until the next redraw. Similarly, return false in on_win will skip the on_line and on_range calls for that window (but any extmarks set in on_win will still be used). A plugin managing multiple sources of decoration should ideally only set one provider, and merge the sources internally. You can use multiple ns_id for the extmarks set/modified inside the callback anyway.

Note: doing anything other than setting extmarks is considered experimental. Doing things like changing options are not explicitly forbidden, but is likely to have unexpected consequences (such as 100% CPU consumption). Doing vim.rpcnotify should be OK, but vim.rpcrequest is quite dubious for the moment.

Note: It is not allowed to remove or update extmarks in on_line or on_range callbacks.

Note: Callbacks will not run for window-hidden, as they are not redrawn.

Attributes:
Lua vim.api only Since: 0.5.0

Parameters:
- {ns_id} (integer) namespace id
- {opts} (vim.api.keyset.set_decoration_provider?) Table of callbacks:
  - on_buf: called for each buffer being redrawn (once per edit, before window callbacks)

    ```
    ["buf", bufnr, tick]
    ```

  - on_end: called at the end of a redraw cycle

    ```
    ["end", tick]
    ```

  - on_line: (deprecated, use on_range instead)

    ```
    ["line", winid, bufnr, row]
    ```

  - on_range: called for each buffer range being redrawn. Range is end-exclusive and may span multiple lines. Range bounds point to the first byte of a character. An end position of the form (lnum, 0), including (number of lines, 0), is valid and indicates that EOL of the preceding line is included.

    ```
    ["range", winid, bufnr, begin_row, begin_col, end_row, end_col]
    ```

    In addition to returning a boolean, it is also allowed to return a skip_row, skip_col pair of integers. This implies that this function does not need to be called until a range which continues beyond the skipped position. A single integer return value skip_row is short for skip_row, 0
  - on_start: called first on each screen redraw

    ```
    ["start", tick]
    ```

  - on_win: called when starting to redraw a specific window.

    ```
    ["win", winid, bufnr, toprow, botrow]
    ```

nvim__ns_get({ns_id}) nvim__ns_get()
WARNING: This feature is experimental/unstable.

Get the properties for namespace

Parameters:
- {ns_id} (integer) Namespace

Return:
(`vim.api.keyset.ns_opts`) Map defining the namespace properties, see nvim__ns_set()

nvim__ns_set({ns_id}, {opts}) nvim__ns_set()
WARNING: This feature is experimental/unstable.

Set some properties for namespace

Parameters:
- `{ns_id}` (`integer`) Namespace
- `{opts}` (`vim.api.keyset.ns_opts?`) Optional parameters to set:
  - wins: a list of windows to be scoped in

Options Functions **api-options**

nvim_get_all_options_info() nvim_get_all_options_info()
Gets the option information for all options.

The dict has the full option names as keys and option metadata dicts as detailed at nvim_get_option_info2().

Attributes:
Since: 0.5.0

Return:
(`table<string,any>`) dict of all options

See also:
- nvim_get_commands()

nvim_get_option_info2({name}, {opts}) nvim_get_option_info2()
Gets the option information for one option from arbitrary buffer or window

Resulting dict has keys:
- name: Name of the option (like 'filetype')
- shortname: Shortened name of the option (like 'ft')
- type: type of option ("string", "number" or "boolean")
- default: The default value for the option
- was_set: Whether the option was set.
- last_set_sid: Last set script id (if any)
- last_set_linenr: line number where option was set
- last_set_chan: Channel where option was set (0 for local)
- scope: one of "global", "win", "buf", or "tab"
- global_local: whether win or buf option has a global value
- commalist: List of comma separated values
- flaglist: List of single char flags

When `{scope}` is not provided, the last set information applies to the local value in the current buffer or window if it is available, otherwise the global value information is returned. This behavior can be disabled by explicitly specifying `{scope}` in the `{opts}` table.

Attributes:
Since: 0.9.0

Parameters:
- `{name}` (`string`) Option name
- `{opts}` (`vim.api.keyset.option?`) Optional parameters
  - buf: Buffer number. Used for getting buffer local options. Implies `{scope}` is "local".
  - scope: One of "global" or "local". Analogous to :setglobal and :setlocal, respectively.
  - win: window-ID. Used for getting window local options.

Return:
(`vim.api.keyset.get_option_info`) Option Information

nvim_get_option_value({name}, {opts}) nvim_get_option_value()
Gets the value of an option. The behavior of this function matches that of :set: the local value of an option is returned if it exists; otherwise, the global value is returned. Local values always correspond to the current buffer or window, unless "buf" or "win" is set in `{opts}`.

Attributes:
Since: 0.7.0

Parameters:
- `{name}` (`string`) Option name
- `{opts}` (`vim.api.keyset.option?`) Optional parameters
  - buf: Buffer number. Used for getting buffer local options. Implies `{scope}` is "local".
  - filetype: filetype. Used to get the default option for a specific filetype. Cannot be used with any other option. **Note:** this will trigger ftplugin and all FileType autocommands for the corresponding filetype.
  - scope: One of "global" or "local". Analogous to :setglobal and :setlocal, respectively.
  - tab: tab-ID for tab-local options. Currently only supports "cmdheight". Tabpage `0` means the current tabpage.
  - win: window-ID. Used for getting window local options.

Return:
(any) Option value

nvim_set_option_value() nvim_set_option_value()
nvim_set_option_value({name}, {value}, {opts}) Sets the value of an option. The behavior of this function matches that of :set: for global-local options, both the global and local value are set unless otherwise specified with `{scope}`.

Note that `win` and `buf` cannot be used together.

Attributes:
Since: 0.7.0

Parameters:
- {name} (string) Option name
- {value} (any) New option value. May be structured for dict/list options, e.g. 'formatoptions' accepts { 'c', 'r' }, 'listchars' accepts { eol = '~' }.
- {opts} (vim.api.keyset.option?) Optional parameters
  - buf: Buffer number. Used for setting buffer local option.
  - dry_run: (boolean?, default: false) If true, validates the option value without setting it.
  - operation: (string?, default: "set") One of "set", "append", "prepend", or "remove". Corresponds to :set=, :set+=, :set^=, and :set-=. With "remove", each item in a structured value is removed regardless of order/adjacency.
  - scope: One of "global" or "local". Analogous to :setglobal and :setlocal, respectively.
  - tab: tab-ID for tab-local options (currently only 'cmdheight'). Tabpage 0 means the current tabpage. If a non-current tab is given, the value will take effect when it is switched-to.
  - win: window-ID. Used for setting window local option.

Return:
(any) Option value

Tabpage Functions

api-tabpage

nvim_open_tabpage({buf}, {enter}, {config}) nvim_open_tabpage()
Opens a new tabpage.

Attributes:
not allowed when textlock is active Since: 0.12.0

Parameters:
- {buf} (integer) Buffer to open in the first window of the new tabpage. Use 0 for current buffer.
- {enter} (boolean) Enter the tabpage (make it the current tabpage).
- {config} (vim.api.keyset.tabpage_config) Configuration for the new tabpage. Keys:
  - after: Position to insert tabpage (default: -1; after current). 0 = first, N = after Nth.

Return:
(integer) tab-ID of the new tabpage

nvim_tabpage_del_var({tabpage}, {name}) nvim_tabpage_del_var()
Removes a tab-scoped (t:) variable

Attributes:
Since: 0.1.0

Parameters:
- {tabpage} (integer) tab-ID, or 0 for current tabpage
- {name} (string) Variable name

nvim_tabpage_get_number({tabpage}) nvim_tabpage_get_number()
Gets the tabpage number

Attributes:
Since: 0.1.0

Parameters:
- {tabpage} (integer) tab-ID, or 0 for current tabpage

Return:
(integer) Tabpage number

nvim_tabpage_get_var({tabpage}, {name}) nvim_tabpage_get_var()
Gets a tab-scoped (t:) variable

Attributes:
Since: 0.1.0

Parameters:
- {tabpage} (integer) tab-ID, or 0 for current tabpage
- {name} (string) Variable name

Return:
(any) Variable value

nvim_tabpage_get_win({tabpage}) nvim_tabpage_get_win()
Gets the current window in a tabpage

Attributes:
Since: 0.1.0

Parameters:
- {tabpage} (integer) tab-ID, or 0 for current tabpage

Return:
(integer) window-ID

nvim_tabpage_is_valid({tabpage}) nvim_tabpage_is_valid()
Checks if a tabpage is valid

Attributes:
Since: 0.1.0

Parameters:
- {tabpage} (integer) tab-ID, or 0 for current tabpage

Return:
(boolean) true if the tabpage is valid, false otherwise

nvim_tabpage_list_wins({tabpage}) nvim_tabpage_list_wins()
Gets the windows in a tabpage

Attributes:
Since: 0.1.0

Parameters:
- {tabpage} (integer) tab-ID, or 0 for current tabpage

Return:
(integer[]) List of windows in tabpage

nvim_tabpage_set_var()
nvim_tabpage_set_var({tabpage}, {name}, {value}) Sets a tab-scoped (t:) variable

Attributes:
Since: 0.1.0

Parameters:
- {tabpage} (integer) tab-ID, or 0 for current tabpage
- {name} (string) Variable name
- {value} (any) Variable value

nvim_tabpage_set_win({tabpage}, {win}) nvim_tabpage_set_win()
Sets the current window in a tabpage

Attributes:
Since: 0.10.0

Parameters:
- {tabpage} (integer) tab-ID, or 0 for current tabpage
- {win} (integer) window-ID, must already belong to {tabpage}

UI Functions api-ui

nvim_ui_attach({width}, {height}, {options}) nvim_ui_attach()
Activates UI events on the channel.

Entry point of all UI clients. Allows --embed to continue startup. Implies that the client is ready to show the UI. Adds the client to the list of UIs. nvim_list_uis()

Note:
- If multiple UI clients are attached, the global screen dimensions degrade to the smallest client. E.g. if client A requests 80x40 but client B requests 200x100, the global screen has size 80x40.

Attributes:
RPC only Since: 0.1.0

Parameters:
- {width} (integer) Requested screen columns
- {height} (integer) Requested screen rows
- {options} (table<string,any>) ui-option map

nvim_ui_detach() nvim_ui_detach()
Deactivates UI events on the channel.

Removes the client from the list of UIs. nvim_list_uis()

Attributes:
RPC only Since: 0.1.0

nvim_ui_pum_set_bounds()
nvim_ui_pum_set_bounds({width}, {height}, {row}, {col}) Tells Nvim the geometry of the popupmenu, to align floating windows with an external popup menu.

Note that this method is not to be confused with nvim_ui_pum_set_height(), which sets the number of visible items in the popup menu, while this function sets the bounding box of the popup menu, including visual elements such as borders and sliders. Floats need not use the same font size, nor be anchored to exact grid corners, so one can set floating-point numbers to the popup menu geometry.

Attributes:
RPC only Since: 0.5.0

Parameters:
- {width} (number) Popupmenu width.
- {height} (number) Popupmenu height.
- {row} (number) Popupmenu row.
- {col} (number) Popupmenu height.

nvim_ui_pum_set_height({height}) nvim_ui_pum_set_height()
Tells Nvim the number of elements displaying in the popupmenu, to decide <PageUp> and <PageDown> movement.

Attributes:
RPC only Since: 0.4.0

Parameters:

- {height} (integer) Popupmenu height, must be greater than zero.

nvim_ui_send({content}) nvim_ui_send()
Sends arbitrary data to a UI. Use this instead of nvim_chan_send() or io.stdout:write(), if you really want to write to the TUI host terminal.

Emits a "ui_send" event to all UIs with the "stdout_tty" ui-option set. UIs are expected to write the received data to a connected TTY if one exists.

Attributes:
Since: 0.12.0

Parameters:
- {content} (string) Content to write to the TTY

nvim_ui_set_focus({gained}) nvim_ui_set_focus()
Tells the nvim server if focus was gained or lost by the GUI

Attributes:
RPC only Since: 0.9.0

Parameters:
- {gained} (boolean)

nvim_ui_set_option({name}, {value}) nvim_ui_set_option()

Attributes:
RPC only Since: 0.1.0

Parameters:
- {name} (string)
- {value} (any)

nvim_ui_try_resize({width}, {height}) nvim_ui_try_resize()

Attributes:
RPC only Since: 0.1.0

Parameters:
- {width} (integer)
- {height} (integer)

nvim_ui_try_resize_grid()
nvim_ui_try_resize_grid({grid}, {width}, {height}) Tell Nvim to resize a grid. Triggers a grid_resize event with the requested grid size or the maximum size if it exceeds size limits.

On invalid grid handle, fails with error.

Attributes:
RPC only Since: 0.4.0

Parameters:
- {grid} (integer) The handle of the grid to be changed.
- {width} (integer) The new requested width.
- {height} (integer) The new requested height.

Win_config Functions api-win_config

nvim_open_win({buf}, {enter}, {config}) nvim_open_win()
Opens a new split window, floating window, or external window.
- Specify relative to create a floating window. Floats are drawn over the split layout, relative to a position in some other window. See api-floatwin.
  - Floats must specify width and height.
- Specify external to create an external window. External windows are displayed as separate top-level windows managed by the ui-multigrid UI (not Nvim).
- If relative and external are omitted, a normal "split" window is created.
  - The win key decides which window to split. If nil or 0, the split will be adjacent to the current window. If -1, a top-level split will be created.
  - Use vertical and split to control split direction. For vertical, the exact direction is determined by 'splitright' and 'splitbelow'.
  - Split windows cannot have bufpos, row, col, border, title, footer.

With relative=editor (row=0,col=0) refers to the top-left corner of the screen-grid and (row=Lines-1,col=Columns-1) refers to the bottom-right corner. Fractional values are allowed, but the builtin implementation (used by non-multigrid UIs) will always round down to nearest integer.

Out-of-bounds values, and configurations that make the float not fit inside the main editor, are allowed. The builtin implementation truncates values so floats are fully within the main screen grid. External GUIs could let floats hover outside of the main window like a tooltip, but this should not be used to specify arbitrary WM screen positions.

Examples:

```
-- Window-relative float with 'statusline' enabled:
local w1 = vim.api.nvim_open_win(0, false,
  {relative='win', row=3, col=3, width=40, height=4})
vim.wo[w1].statusline = vim.o.statusline

-- Buffer-relative float (travels as buffer is scrolled):
vim.api.nvim_open_win(0, false,
  {relative='win', width=40, height=4, bufpos={100,10}})

-- Vertical split left of the current window:
vim.api.nvim_open_win(0, false, { split = 'left', win = 0, })
```

Attributes:
not allowed when textlock is active Since: 0.4.0

Parameters:

- {buf} (integer) Buffer to display, or 0 for current buffer
- {enter} (boolean) Enter the window (make it the current window)
- {config} (vim.api.keyset.win_config) Map defining the window configuration. Keys:
  - anchor: Decides which corner of the float to place at (row,col):
    - "NW" northwest (default)
    - "NE" northeast
    - "SW" southwest
    - "SE" southeast
  - border: (string|string[], default: 'winborder' option) Window border. The string form accepts the same values as the 'winborder' option. The array form must have a length of eight or any divisor of eight, specifying the chars that form the border in a clockwise fashion starting from the top-left corner. For example, the double-box style can be specified as:

    ```
    [ "╔", "═" ,"╗", "║", "╝", "═", "╚", "║" ].
    ```

    If fewer than eight chars are given, they will be repeated. An ASCII border could be specified as:

    ```
    [ "/", "-", \"\\\\\", "|" ],
    ```

    Or one char for all sides:

    ```
    [ "x" ].
    ```

    Empty string can be used to hide a specific border. This example will show only vertical borders, not horizontal:

    ```
    [ "", "", "", ">", "", "", "", "<" ]
    ```

    By default, hl-FloatBorder highlight is used, which links to hl-WinSeparator when not defined. Each border side can specify an optional highlight:

    ```
    [ ["+", "MyCorner"], ["x", "MyBorder"] ].
    ```

  - bufpos: Places float relative to buffer text (only when relative="win"). Takes a tuple of zero-indexed [line, column]. row and col if given are applied relative to this position, else they default to:
    - row=1 and col=0 if anchor is "NW" or "NE"
    - row=0 and col=0 if anchor is "SW" or "SE" (thus like a tooltip near the buffer text).
  - col: Column position in units of screen cell width, may be fractional.
  - external: GUI should display the window as an external top-level window. Currently accepts no other positioning configuration together with this.
  - fixed: If true when anchor is NW or SW, the float window would be kept fixed even if the window would be truncated.
  - focusable: (default: true) Enable focus by user actions (wincmds, mouse events). Non-|focusable| windows can be entered by nvim_set_current_win(), or by mouse events if mouse=true.
  - footer: (string|string[]?) Footer in window border: string or list of [text, highlight] tuples. Default highlight group is FloatFooter.
  - footer_pos: (default: "left") Footer position. One of: "left", "center", "right". The footer field must be set also.
  - height: Window height (in character cells). Minimum of 1.
  - hide: Hides the floating window. window-hidden
  - mouse: (default: focusable value) If true, mouse events interact with the window normally; if false, they pass through to the window behind it.
  - noautocmd: Block all autocommands for the duration of the call. Cannot be changed by nvim_win_set_config().
  - relative: Sets the window layout to "floating", placed at (row,col) coordinates relative to:
    - "cursor" Cursor position in current window.
    - "editor" The global editor grid.
    - "laststatus" 'laststatus' if present, or last row.
    - "mouse" Mouse position.
    - "tabline" Tabline if present, or first row.
    - "win" Window given by the win field, or current window.
  - row: Row position in units of "screen cell height", may be fractional.
  - split: Split direction: "left", "right", "above", "below".
  - style: (optional) Configure the appearance of the window:
    - "" No special style.
    - "minimal" Nvim will display the window with many UI options disabled. This is useful when displaying a temporary float where the text should not be edited. Disables 'number', 'relativenumber', 'cursorline', 'cursorcolumn', 'foldcolumn', 'spell' and 'list' options. 'signcolumn' is changed to auto and 'colorcolumn' is cleared. 'statuscolumn' is changed to empty. The end-of-buffer region is hidden by setting eob flag of 'fillchars' to a space char, and clearing the hl-EndOfBuffer region in 'winhighlight'.

- title: (optional) Title in window border, string or list. List should consist of `[text, highlight]` tuples. If string, or a tuple lacks a highlight, the default highlight group is `FloatTitle`.
- title_pos: (default: "left") Title position. One of: "left", "center", "right". The `title` field must be set also.
- vertical: Split vertically :vertical.
- width: Window width (in character cells). Minimum of 1.
- win: window-ID target window. Can be in a different tab page. Determines the window to split (negative values act like :topleft, :botright), the relative window for a `relative="win"` float, or just the target tab page (inferred from the window) for others.
- zindex: (positive integer, default: 50) Stacking order. Floats with higher `zindex` overlay floats with lower indices. Below 100 is recommended, unless there is a good reason to overshadow builtin elements. The cursor is dimmed if an unfocused float above the cursor exceeds the zindex of the current window by 50. These screen elements have hard-coded z-indices:
  - 100: ins-completion-menu popupmenu
  - 200: message scrollback (pager)
  - 250: cmdline-completion popupmenu (wildoptions=pum)
- _cmdline_offset: (EXPERIMENTAL) When provided, anchor the cmdline-completion popupmenu to this window, with an offset in screen cell width.

Return:
(`integer`) window-ID, or 0 on error

nvim_win_get_config({win}) nvim_win_get_config()
Gets window config as a dict which can be passed to nvim_open_win() as the `config` parameter.

For non-floating windows, `relative` is empty, thus you can check that field to detect if a window is a floatwin:

```
vim.print(vim.api.nvim_win_get_config(0).relative == '' and 'non-float' or 'float')
-- Or use win_gettype().
vim.print(vim.fn.win_gettype())
```

Attributes:
Since: 0.4.0

Parameters:
- {win} (`integer`) window-ID, or 0 for current window

Return:
(`vim.api.keyset.win_config`) Map defining the window configuration, see nvim_open_win()

nvim_win_set_config({win}, {config}) nvim_win_set_config()
Reconfigures the layout and properties of a window.
- Updates only the given keys; unspecified (`nil`) keys will not be changed.
- Can move a window to another tabpage.
- Can transform a window to/from a float.
- Keys `row` / `col` / `relative` must be specified together.
- Cannot move the last window in a tabpage to a different one.

Example: to convert a floating window to a "normal" split window, specify the `win` field:

```
vim.api.nvim_win_set_config(0, { split = 'above', win = vim.fn.win_getid(1), })
```

Attributes:
Since: 0.4.0

Parameters:
- {win} (`integer`) window-ID, or 0 for current window
- {config} (`vim.api.keyset.win_config`) Map defining the window configuration, see nvim_open_win()

See also:
- nvim_open_win()

Window Functions api-window

nvim_win_call({win}, {fn}) nvim_win_call()
Calls function `fn` in the context of window `win` and returns its result (may be multiple values).

Attributes:
Lua vim.api only Since: 0.5.0

Parameters:
- {win} (`integer`) window-ID, or 0 for current window.
- {fn} (`function`) Lua function to call inside the window.

Return:
(any) Value(s) returned by `fn()`.

See also:
- win_execute()
- nvim_buf_call()

nvim_win_close({win}, {force}) nvim_win_close()
Closes the window (like :close with a window-ID).

Attributes:
not allowed when textlock is active Since: 0.4.0

Parameters:
- {win} (`integer`) window-ID, or 0 for current window

- {force} (boolean) Behave like :close! The last window of a buffer with unwritten changes can be closed. The buffer will become hidden, even if 'hidden' is not set.

nvim_win_del_var({win}, {name}) nvim_win_del_var()
Removes a window-scoped (w:) variable

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window
- {name} (string) Variable name

nvim_win_get_buf({win}) nvim_win_get_buf()
Gets the current buffer in a window

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window

Return:
(integer) Buffer id

nvim_win_get_cursor({win}) nvim_win_get_cursor()
Gets the (1,0)-indexed, buffer-relative cursor position for a given window (different windows showing the same buffer have independent cursor positions). api-indexing

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window

Return:
([integer, integer]) (row, col) tuple

See also:
- getcurpos()

nvim_win_get_height({win}) nvim_win_get_height()
Gets the window height

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window

Return:
(integer) Height as a count of rows

nvim_win_get_number({win}) nvim_win_get_number()
Gets the window number

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window

Return:
(integer) Window number

nvim_win_get_position({win}) nvim_win_get_position()
Gets the window position in display cells. First position is zero.

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window

Return:
([integer, integer]) (row, col) tuple with the window position

nvim_win_get_tabpage({win}) nvim_win_get_tabpage()
Gets the window tabpage

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window

Return:
(integer) Tabpage that contains the window

nvim_win_get_var({win}, {name}) nvim_win_get_var()
Gets a window-scoped (w:) variable

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window
- {name} (string) Variable name

Return:
(any) Variable value

nvim_win_get_width({win}) nvim_win_get_width()
Gets the window width

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window

Return:
(integer) Width as a count of columns

nvim_win_hide({win}) nvim_win_hide()
Closes the window and hides the buffer it contains (like :hide with a window-ID; unrelated to the hide flag of nvim_open_win(), nvim_win_get_config()).

Compare :close and nvim_win_close(), which close the buffer instead of hiding it.

Attributes:
not allowed when textlock is active Since: 0.5.0

Parameters:
- {win} (integer) window-ID, or 0 for current window

nvim_win_is_valid({win}) nvim_win_is_valid()
Checks if a window is valid

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window

Return:
(boolean) true if the window is valid, false otherwise

nvim_win_resize({win}, {width}, {height}, {opts}) nvim_win_resize()
Resizes a window, anchored to a given edge.

The window first takes space from the non-anchored side, and only then from the anchored side. The "anchor" decides the fixed edge, so a window can grow or shrink in any direction.

Can set height, width (only for a vertical split), or both. The "anchor" applies to the matching axis (top/bottom for height, left/right for width); the other axis uses its default anchor.

Attributes:
Since: 0.13.0

Parameters:
- {win} (integer) window-ID, or 0 for current window
- {width} (integer) New width (columns), or -1 for "no change".
- {height} (integer) New height (rows), or -1 for "no change".
- {opts} (vim.api.keyset.win_resize?) Optional parameters.
  - anchor: Edge that stays fixed while the opposite edge moves; the neighbor on the moving side is resized first. One of:
    - "top" (default) or "bottom"
    - "left" (default) or "right"

nvim_win_set_buf({win}, {buf}) nvim_win_set_buf()
Sets the current buffer in a window.

Note: As a side-effect, this executes BufEnter and BufLeave autocommands.

Attributes:
not allowed when textlock is active Since: 0.3.2

Parameters:
- {win} (integer) window-ID, or 0 for current window
- {buf} (integer) Buffer id

nvim_win_set_cursor({win}, {pos}) nvim_win_set_cursor()
Sets the (1,0)-indexed cursor position (byte offset) in the window. api-indexing This scrolls the window even if it is not the current one.

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window
- {pos} ([integer, integer]) (row, col) tuple representing the new position

nvim_win_set_hl_ns({win}, {ns_id}) nvim_win_set_hl_ns()
Set highlight namespace for a window. This will use highlights defined with nvim_set_hl() for this namespace, but fall back to global highlights (ns=0) when missing.

This takes precedence over the 'winhighlight' option.

Attributes:

Since: 0.8.0

Parameters:
- {win} (integer)
- {ns_id} (integer) the namespace to use

nvim_win_set_var({win}, {name}, {value}) nvim_win_set_var()
Sets a window-scoped (w:) variable

Attributes:
Since: 0.1.0

Parameters:
- {win} (integer) window-ID, or 0 for current window
- {name} (string) Variable name
- {value} (any) Variable value

nvim_win_text_height({win}, {opts}) nvim_win_text_height()
Computes the number of screen lines occupied by a range of text in a given window. Works for off-screen text and takes folds into account.

Diff filler or virtual lines above a line are counted as a part of that line, unless the line is on "start_row" and "start_vcol" is specified.

Diff filler or virtual lines below the last buffer line are counted in the result when "end_row" is omitted.

Line indexing is similar to nvim_buf_get_text().

Attributes:
Since: 0.10.0

Parameters:
- {win} (integer) window-ID, or 0 for current window.
- {opts} (vim.api.keyset.win_text_height?) Optional parameters:
  - end_row: Ending line index, 0-based inclusive. When omitted end at the very bottom.
  - end_vcol: Ending virtual column index on "end_row", 0-based exclusive, rounded up to full screen lines. When 0 only include diff filler and virtual lines above "end_row". When omitted include the whole line.
  - max_height: Don't add the height of lines below the row for which this height is reached. Useful to e.g. limit the height to the window height, avoiding unnecessary work. Or to find out how many buffer lines beyond "start_row" take up a certain number of logical lines (returned in "end_row" and "end_vcol").
  - start_row: Starting line index, 0-based inclusive. When omitted start at the very top.
  - start_vcol: Starting virtual column index on "start_row", 0-based inclusive, rounded down to full screen lines. When omitted include the whole line.

Return:
(vim.api.keyset.win_text_height_ret) Dict containing text height information, with these keys:
- all: The total number of screen lines occupied by the range.
- fill: The number of diff filler or virtual lines among them.
- end_row: The row on which the returned height is reached (first row of a closed fold).
- end_vcol: Ending virtual column in "end_row" where "max_height" or the returned height is reached. 0 if "end_row" is a closed fold.

See also:
- virtcol() for text width.

Automatic commands autocommand

For a basic explanation, see section 40.3 in the user manual.

1. Introduction autocmd-intro

You can specify commands to be executed automatically when reading or writing a file, when entering or leaving a buffer or window, and when exiting Vim. For example, you can create an autocommand to set the 'cindent' option for files matching *.c. You can also use autocommands to implement advanced features, such as editing compressed files (see gzip-example). The usual place to put autocommands is in your vimrc file.

E203 E204 E143 E855 E937 E952 E1312

WARNING: Using autocommands is very powerful, and may lead to unexpected side effects. Be careful not to destroy your text.

- It's a good idea to do some testing on an expendable copy of a file first.

  For example: If you use autocommands to decompress a file when starting to edit it, make sure that the autocommands for compressing when writing work correctly.
- Be prepared for an error halfway through (e.g., disk full). Vim will mostly

  be able to undo the changes to the buffer, but you may have to clean up the changes to other files by hand (e.g., compress a file that has been decompressed).
- If the BufRead* events allow you to edit a compressed file, the FileRead*

  events should do the same (this makes recovery possible in some rare cases). It's a good idea to use the same autocommands for the File* and Buf* events when possible.

2. Defining autocommands autocmd-define

Use nvim_create_autocmd() to define an event handler from Lua.

Vimscript commands are described below.

:au :autocmd

:au[tocmd] [group] {event} {aupat} [++once] [++nested] {cmd}
Add {cmd} to the list of commands that Vim will execute automatically on {event} for a file matching {aupat} autocmd-pattern.
Here {event} cannot be "*". E1155
Note: A quote character is seen as argument to the :autocmd and won't start a comment.
Nvim always adds {cmd} after existing autocommands so they execute in the order in which they were defined.
See autocmd-nested for [++nested].

autocmd-once

If [++once] is supplied the command is executed once (per {event}, which may be a comma-separated list), then removed ("one shot").

The special pattern <buffer> or <buffer=N> defines a buffer-local autocommand. See autocmd-buflocal.

Note: The ":autocmd" command can only be followed by another command when the "|" appears where the pattern is expected. This works:

```
:augroup mine | au! BufRead | augroup END
```

But this sees "augroup" as part of the defined command:

```
:augroup mine | au! BufRead * | augroup END
:augroup mine | au BufRead * set tw=70 | augroup END
```

Instead you can put the group name into the command:

```
:au! mine BufRead *
:au mine BufRead * set tw=70
```

Or use :execute:

```
:augroup mine | exe "au! BufRead *" | augroup END
:augroup mine | exe "au BufRead * set tw=70" | augroup END
```

autocmd-expand

Note that special characters (e.g., "%", "<cword>") in the ":autocmd" arguments are not expanded when the autocommand is defined. These will be expanded when the Event is recognized, and the {cmd} is executed. The only exception is that "<sfile>" (unlike "<script>") is expanded when the autocmd is defined. Example:

```
:au BufNewFile,BufRead *.html so <sfile>:h/html.vim
```

Here Vim expands <sfile> to the name of the file containing this line.
However, <sfile> works differently in a function, in which case it's better to
use :execute with <script> to achieve the same purpose:

```
:exe $'au BufNewFile,BufRead *.html so {expand("<script>:h")}/html.vim'
```

:autocmd adds to the list of autocommands regardless of whether they are
already present. When your .vimrc file is sourced twice, the autocommands
will appear twice. To avoid this, define your autocommands in a group, so
that you can easily clear them:

```
augroup vimrc
  " Remove all vimrc autocommands
  autocmd!
  au BufNewFile,BufRead *.html so <sfile>:h/html.vim
augroup END
```

If you don't want to remove all autocommands, you can instead use a variable
to ensure that Vim includes the autocommands only once:

```
:if !exists("autocommands_loaded")
:  let autocommands_loaded = 1
:  au ...
:endif
```

When the [group] argument is not given, Vim uses the current group (as defined
with ":augroup"); otherwise, Vim uses the group defined with [group]. Note
that [group] must have been defined before. You cannot define a new group
with ":au group ..."; use ":augroup" for that.

While testing autocommands, you might find the 'verbose' option to be useful:

```
:set verbose=9
```

This setting makes Vim echo the autocommands as it executes them.

When defining an autocommand in a script, it will be able to call functions
local to the script and use mappings local to the script. When the event is
triggered and the command executed, it will run in the context of the script
it was defined in. This matters if <SID> is used in a command.

When executing the commands, the message from one command overwrites a
previous message. This is different from when executing the commands
manually. Mostly the screen will not scroll up, thus there is no hit-enter
prompt. When one command outputs two messages this can happen anyway.

3. Removing autocommands

autocmd! **autocmd-remove**

Using Lua, these functions remove autocmds and autocmd groups:

- nvim_del_autocmd()
- nvim_del_augroup_by_id()
- nvim_del_augroup_by_name()

Vimscript commands are described below.

:au[tocmd]! [group] {event} {aupat} [++once] [++nested] {cmd}
Remove all autocommands associated with {event} and
{aupat}, and add the command {cmd}.
See autocmd-once for [++once].
See autocmd-nested for [++nested].

:au[tocmd]! [group] {event} {aupat}
Remove all autocommands associated with {event} and
{aupat}.

:au[tocmd]! [group] * {aupat}
Remove all autocommands associated with {aupat} for
all events.

:au[tocmd]! [group] {event}
Remove ALL autocommands for {event}.
Warning: You should not do this without a group for
BufRead and other common events, it can break
plugins, syntax highlighting, etc.

:au[tocmd]! [group]
Remove ALL autocommands.
Note: a quote will be seen as argument to the :autocmd
and won't start a comment.
Warning: You should normally not do this without a
group, it breaks plugins, syntax highlighting, etc.

When the [group] argument is not given, Vim uses the current group (as defined with ":augroup"); otherwise, Vim uses the group defined with [group].

4. Listing autocommands

autocmd-list

:au[tocmd] [group] {event} {aupat}
Show the autocommands associated with {event} and {aupat}.

:au[tocmd] [group] * {aupat}
Show the autocommands associated with {aupat} for all events.

:au[tocmd] [group] {event}
Show all autocommands for {event}.

:au[tocmd] [group] Show all autocommands.

If you provide the [group] argument, Vim lists only the autocommands for [group]; otherwise, Vim lists the autocommands for ALL groups. Note that this argument behavior differs from that for defining and removing autocommands.

In order to list buffer-local autocommands, use a pattern in the form <buffer> or <buffer=N>. See autocmd-buflocal.

:autocmd-verbose

When 'verbose' is non-zero, listing an autocommand will also display where it was last defined. Example:

```
:verbose autocmd BufEnter
FileExplorer  BufEnter
    *     call s:LocalBrowse(expand("<amatch>"))
        Last set from /usr/share/vim/vim-7.0/plugin/NetrwPlugin.vim
```

See :verbose-cmd for more information.

5. Events

autocmd-events E215 E216

You can specify a comma-separated list of event names. No white space can be used in this list. The command applies to all the events in the list.

For READING FILES there are four kinds of events possible:

BufNewFile		starting to edit a non-existent file
BufReadPre	BufReadPost	starting to edit an existing file
FilterReadPre	FilterReadPost	read the temp file with filter output
FileReadPre	FileReadPost	any other file read

Vim uses only one of these four kinds when reading a file. The "Pre" and "Post" events are both triggered, before and after reading the file.

Note that the autocommands for the "*ReadPre" events and all the Filter events are not allowed to change the current buffer (you will get an error message if this happens). This is to prevent the file to be read into the wrong buffer.

Note that the 'modified' flag is reset AFTER executing the BufReadPost and BufNewFile autocommands. But when the 'modified' option was set by the autocommands, this doesn't happen.

You can use the 'eventignore' option to ignore a number of events or all events.

event-data-types

Events that provide event-data have a corresponding vim.event.<name>.data type defined in runtime/lua/vim/_meta/events.lua. If you have Lua LSP enabled, you can annotate your handler to get type-checking and completion:

```
vim.api.nvim_create_autocmd('LspAttach', {
  ---@param ev {data: vim.event.lspattach.data}
  callback = function(ev)
    local client = vim.lsp.get_client_by_id(ev.data.client_id)
  end,
})
```

events {event}

Nvim recognizes the following events. Names are case-insensitive.

BufAdd

BufAdd After adding a new buffer or existing unlisted buffer to the buffer list (except during startup, see VimEnter), or renaming a listed buffer.
Before BufEnter.
NOTE: Current buffer "%" is not the target buffer "<afile>", "<abuf>". <buffer=abuf>

BufDelete

BufDelete Before deleting a buffer from the buffer list. The BufUnload may be called first (if the buffer was loaded).
Also used just before a buffer in the buffer list is renamed.
NOTE: Current buffer "%" is not the target

buffer "<afile>", "<abuf>". <buffer=abuf>
Do not change to another buffer.

BufEnter

BufEnter After entering (visiting, switching-to) a new or existing buffer. Useful for setting filetype options. Compare BufNew which does not trigger for existing buffers.
After BufAdd.
After BufReadPost.

BufFilePost

BufFilePost After changing the name of the current buffer with the ":file" or ":saveas" command.

BufFilePre

BufFilePre Before changing the name of the current buffer with the ":file" or ":saveas" command.

BufHidden

BufHidden Before a buffer becomes hidden: when there are no longer windows that show the buffer, but the buffer is not unloaded or deleted.

Not used for ":qa" or ":q" when exiting Vim.
NOTE: Current buffer "%" is not the target buffer "<afile>", "<abuf>". <buffer=abuf>

BufLeave

BufLeave Before leaving to another buffer. Also when leaving or closing the current window and the new current window is not for the same buffer.

Not used for ":qa" or ":q" when exiting Vim.

BufNew

BufNew After creating a new buffer (except during startup, see VimEnter) or renaming an existing buffer. Unlike BufEnter, visiting (switching to) an existing buffer will not trigger this again.
NOTE: Current buffer "%" is not the target buffer "<afile>", "<abuf>". <buffer=abuf>
See also BufAdd, BufNewFile.

BufNewFile

BufNewFile When starting to edit a file that doesn't exist. Can be used to read in a skeleton file.

BufRead BufReadPost

BufRead or BufReadPost When starting to edit a new buffer, after reading the file into the buffer, before processing modelines. See BufWinEnter to do something after processing modelines.
Also triggered:

- when writing an unnamed buffer in a way that

the buffer gets a name

- after successfully recovering a file

- for the "filetypedetect" group when

executing ":filetype detect"
Not triggered:

- for the :read file command

- when the file doesn't exist

BufReadCmd

BufReadCmd Before starting to edit a new buffer. Should read the file into the buffer. Cmd-event

BufReadPre E200 E201

BufReadPre When starting to edit a new buffer, before reading the file into the buffer. Not used if the file doesn't exist.

BufUnload

BufUnload Before unloading a buffer, when the text in the buffer is going to be freed.
After BufWritePost.

Before BufDelete.
Triggers for all loaded buffers when Vim is
going to exit.
NOTE: Current buffer "%" is not the target
buffer "<afile>", "<abuf>". <buffer=abuf>

E1546

Do not switch buffers or windows!
Not triggered when exiting and v:dying is 2 or
more.

BufWinEnter

BufWinEnter After a buffer is displayed in a window. This
may be when the buffer is loaded (after
processing modelines) or when a hidden buffer
is displayed (and is no longer hidden).

Not triggered for :split without arguments,
since the buffer does not change, or :split
with a file already open in a window.
Triggered for ":split" with the name of the
current buffer, since it reloads that buffer.

BufWinLeave

BufWinLeave Before a buffer is removed from a window.
Not when it's still visible in another window.
Also triggered when exiting.
Before BufUnload, BufHidden.
NOTE: Current buffer "%" is not the target
buffer "<afile>", "<abuf>". <buffer=abuf>
Not triggered when exiting and v:dying is 2 or
more.

BufWipeout

BufWipeout Before completely deleting a buffer. The
BufUnload and BufDelete events may be called
first (if the buffer was loaded and was in the
buffer list). Also used just before a buffer
is renamed (also when it's not in the buffer
list).
NOTE: Current buffer "%" is not the target
buffer "<afile>", "<abuf>". <buffer=abuf>
Do not change to another buffer.

BufWrite BufWritePre

BufWrite or BufWritePre Before writing the whole buffer to a file.

BufWriteCmd

BufWriteCmd Before writing the whole buffer to a file.
Should do the writing of the file and reset
'modified' if successful, unless '+' is in
'cpo' and writing to another file cpo-+.
The buffer contents should not be changed.
When the command resets 'modified' the undo
information is adjusted to mark older undo
states as 'modified', like :write does. Use
the '[and '] marks for the range of lines.
Cmd-event

BufWritePost

BufWritePost After writing the whole buffer to a file
(should undo the commands for BufWritePre).

ChanClose

ChanClose After a channel was closed, before it is removed.
This is triggered even when inside an
autocommand defined without autocmd-nested.
Sets these v:event keys:
    info   as from nvim_get_chan_info()

ChanInfo

ChanInfo State of channel changed, for instance the
client of a RPC channel described itself.
This is triggered even when inside an
autocommand defined without autocmd-nested.
Sets these v:event keys:
    info   as from nvim_get_chan_info()

ChanOpen

ChanOpen Just after a channel was opened.
This is triggered even when inside an
autocommand defined without autocmd-nested.
Sets these v:event keys:
    info   as from nvim_get_chan_info()

CmdAtom

CmdAtom After a user action (an "atom" of input): any

normal command, motion, operator, insert session, Visual sequence, Ex cmdline (":cnext<CR>"), mapping (and subatoms), macro, or scroll/mouse.

<amatch> is the atom "type" (see below). Event is deferred.

Only for user input, not programmatic input:

INPUT	ATOM
Typed keys	yes
Mapping, macro "@q"	yes
nvim_input()	yes
nvim_feedkeys()	with "t" (see note)
Hydrogen	yes
:normal	no
API requests	no
"@q" fed by a script	no
multicursor cascade	no
Aborted operation	no
Terminal-mode keys	no
Mouse drag/release	no (TODO?)

Note: All keys fed during a mapping execution are part of the resolved keys. But entering Terminal-mode (e.g. a picker/fuzzy-finder UI) completes the atom: terminal input is not captured, because there is no defined way to delimit the "end" of such an action.

Plugins should not publish "fake" CmdAtom events via nvim_exec_autocmds.

The event-data has these fields (type: vim.event.cmdatom.data):

- atoms: Ordered "subatoms" of a composite atom (complex mapping, Visual sequence).
- changed: Buffer was changed by the command.
- cmd: Command/motion/object name: "w", "f", "iw", "gJ", "<Home>", …. key-notation
- cmdarg: Operand of cmd: the "x" of "fx", the replacement char of r. key-notation
- count: Effective count, or nil.
- keys: Resolved (post-mapping) keysequence. Raw internal bytes (use keytrans() to get key-notation): feed to nvim_feedkeys() (mode "n") to replay. Empty: unreplayable. Nil: lossy capture, replay lhs (mode "m").
- lhs: User input, before it is resolved. Mapping LHS plus any payload it read (getchar()), macro register ("@q"), or Visual sequence. Raw bytes, like keys.
- motionforce forced-motion: "v", "V", or "<C-V>" (key-notation).
- moved: Cursor moved. May be false even if the action is classified as a "motion" (e.g. "w" at buffer end, failed "fx").
- operator: Operator: "d", "g@", "zf", …. key-notation, like cmd.
- pos: Start position: [row, col], 1-indexed row, 0-indexed col.
- reg: Register name.
- text: Payload: full text of an insert session, Ex cmdline, or search. Raw bytes, like keys.
  Examples:

    - typed "iab<Esc>" → text="ab"
    - typed ":cnext<CR>" → text="cnext"
    - typed "<Cmd>cnext<CR>" → text="cnext"
    - typed "iab<Left>c<Esc>" → text="c"
- type:
    - "excmd": Ex cmdline session.
    - "insert": Insert session.
    - "jump": Absolute motion ("G", marks, "H", jumplist).
    - "mapping": Composite (≥2 subatoms).
    - "motion": Cursor-relative motion ("w", "%", "fx", "*").
    - "mouse"
    - "normal": Other commands ("u", CTRL-R, "za", CTRL-W, …).
    - "operator"
    - "scroll": Viewport scroll.
    - "visual": Visual composite.
- undoseq: Undo state after the action (undotree() seq_cur). Decreases on undo.

CmdlineChanged

CmdlineChanged — After EVERY change inside command line. Also triggered during mappings! Use <Cmd> instead of ":" in mappings, to avoid that.
<afile> expands to the cmdline-char.

CmdlineEnter

CmdlineEnter — After entering the command-line (including non-interactive use of ":" in a mapping: use <Cmd> instead to avoid this).
The pattern is matched against cmdline-char.
<afile> expands to the cmdline-char.
Sets these v:event keys:
    cmdlevel
    cmdtype

CmdlineLeave

CmdlineLeave — Before leaving the command-line (including non-interactive use of ":" in a mapping: use <Cmd> instead to avoid this).
<afile> expands to the cmdline-char.
Sets the v:char to the key that exited the command-line (e.g. <CR>, <CTRL-C>, <Esc>).
Sets these v:event keys:
    abort (mutable)
    cmdlevel
    cmdtype
Note: abort can only be changed from false to true: cannot execute an already aborted cmdline by changing it to false.

CmdlineLeavePre

CmdlineLeavePre — Just before leaving the command line, and before CmdlineLeave. Useful for capturing

completion info with cmdcomplete_info(), as this information is cleared before CmdlineLeave is triggered. Triggered for non-interactive use of ":" in a mapping, but not when using <Cmd>. Also triggered when abandoning the command line by typing CTRL-C or <Esc>. <afile> is set to cmdline-char. Sets v:char as with CmdlineLeave.

CmdUndefined

CmdUndefined — When a user command is used but it isn't defined. Useful for defining a command only when it's used. The pattern is matched against the command name. Both <amatch> and <afile> expand to the command name.
This is triggered even when inside an autocommand defined without autocmd-nested.
NOTE: Autocompletion won't work until the command is defined. An alternative is to always define the user command and have it invoke an autoloaded function. See autoload.

CmdwinEnter

CmdwinEnter — After entering the command-line window. Useful for setting options specifically for this special type of window.
<afile> expands to a single character, indicating the type of command-line.
cmdwin-char

CmdwinLeave

CmdwinLeave — Before leaving the command-line window. Useful to clean up any global setting done with CmdwinEnter.
<afile> expands to a single character, indicating the type of command-line.
cmdwin-char

ColorScheme

ColorScheme — After loading a color scheme. :colorscheme
Not triggered if the color scheme is not found.
The pattern is matched against the colorscheme name. <afile> can be used for the name of the actual file where this option was set, and <amatch> for the new colorscheme name.

ColorSchemePre

ColorSchemePre — Before loading a color scheme. :colorscheme
Useful to setup removing things added by a color scheme, before another one is loaded.

CompleteChanged

CompleteChanged

After each time the Insert mode completion menu changed. Not fired on popup menu hide, use CompleteDonePre or CompleteDone for that.

Sets these v:event keys:

completed_item	See complete-items.
height	nr of items visible
width	screen cells
row	top screen row
col	leftmost screen column
size	total nr of items
scrollbar	TRUE if visible

Non-recursive (event cannot trigger itself). Cannot change the text. textlock

The size and position of the popup are also available by calling pum_getpos().

CompleteDone

CompleteDone — After Insert mode completion is done. Either when something was completed or discarded.
ins-completion
complete_info() is cleared before this; use CompleteDonePre if you need it.
v:completed_item gives the completed item, or empty dict if completion was discarded.

Sets these v:event keys:

complete_word	The word that was selected, empty if completion was abandoned (discarded).
complete_type	complete_info_mode

reason Reason for completion being done. Can be one of:

- "accept": completion was accepted by complete_CTRL-Y.
- "cancel": completion was stopped by complete_CTRL-E.
- "discard": completion was abandoned for other reason.

CompleteDonePre

CompleteDonePre After Insert mode completion is done. Either when something was completed or discarded. ins-completion complete_info() is valid during this event. v:completed_item gives the completed item.

CursorHold

CursorHold When there is no user input for 'updatetime' duration, in Normal-mode. Not triggered while waiting for a command argument or movement after an operator, nor while recording a macro. See CursorHold-example.

Note: Interactive commands cannot be used for this event. There is no hit-enter prompt, the screen is updated directly (when needed).
Note: In the future there will probably be another option to set the time.
Hint: to force an update of the status lines use:

```
:let &ro = &ro
```

CursorHoldI

CursorHoldI Like CursorHold, but in Insert mode. Not triggered when waiting for another key, e.g. after CTRL-V, and not in CTRL-X mode insert_expand.

CursorMoved

CursorMoved After the cursor was moved in Normal or Visual mode or to another window. Also when the text of the cursor line has been changed, e.g. with "x", "rx" or "p".
Not always triggered when there is typeahead, while executing commands in a script file, or when an operator is pending.
For an example see match-parens.
Note: Cannot be skipped with :noautocmd.
Careful: This is triggered very often, don't do anything that the user does not expect or that is slow.

CursorMovedC

CursorMovedC After the cursor was moved in the command line. Be careful not to mess up the command line, it may cause Vim to lock up.
<afile> expands to the cmdline-char.

CursorMovedI

CursorMovedI After the cursor was moved in Insert mode.
Not triggered when the popup menu is visible.
Otherwise the same as CursorMoved.

DiffUpdated

DiffUpdated After diffs have been updated. Depending on what kind of diff is being used (internal or external) this can be triggered on every change or when doing :diffupdate.
It is not allowed to change the window layout.

DirChanged

DirChanged After the current-directory was changed.
The pattern can be:
"auto" triggered on 'autochdir'.
"buffer" triggered on :bcd.
"global" triggered on :cd.
"tabpage" triggered on :tcd.
"window" triggered on :lcd.
Sets these v:event keys:

cwd: current working directory
scope: "global", "tabpage", "window", "buffer"
changed_window: v:true if we fired the event switching window (or tab)
<afile> is set to the new directory name.
Non-recursive (event cannot trigger itself).

DirChangedPre

DirChangedPre When the current-directory is going to be changed, as with DirChanged.
The pattern is like with DirChanged.
Sets these v:event keys:
directory: new working directory
scope: "global", "tabpage", "window", "buffer"
changed_window: v:true if we fired the event switching window (or tab)
<afile> is set to the new directory name.
Non-recursive (event cannot trigger itself).

ExitPre

ExitPre When using :quit, :wq in a way it makes Vim exit, or using :qall, just after QuitPre. Can be used to close any non-essential window. Exiting may still be cancelled if there is a modified buffer that isn't automatically saved, use VimLeavePre for really exiting.
See also QuitPre, WinClosed.

FileAppendCmd

FileAppendCmd Before appending to a file. Should do the appending to the file. Use the '[and '] marks for the range of lines. Cmd-event

FileAppendPost

FileAppendPost After appending to a file.

FileAppendPre

FileAppendPre Before appending to a file. Use the '[and '] marks for the range of lines.

FileChangedRO

FileChangedRO Before making the first change to a read-only file. Can be used to checkout the file from a source control system. Not triggered when the change was caused by an autocommand. Triggered when making the first change in a buffer or the first change after 'readonly' was set, just before the change is applied to the text.
WARNING: If the autocommand moves the cursor the effect of the change is undefined.

E788

Cannot switch buffers. You can reload the buffer but not edit another one.

E881

If the number of lines changes saving for undo may fail and the change will be aborted.

FileChangedShell

FileChangedShell When Vim notices that the modification time of a file has changed since editing started.
Also when the file attributes of the file change or when the size of the file changes.
timestamp
Triggered for each changed file, after:

- executing a shell command or :terminal job
- :checktime
- FocusGained

Not used when 'autoread' is set and the buffer was not changed. If a FileChangedShell autocommand exists the warning message and prompt is not given.
v:fcs_reason indicates what happened. Set v:fcs_choice to control what happens next.
NOTE: Current buffer "%" is not the target buffer "<afile>" and "<abuf>". <buffer=abuf>

E246 E811

Cannot switch, jump to or delete buffers.
Non-recursive (event cannot trigger itself).

FileChangedShellPost

FileChangedShellPost After handling a file that was changed outside of Vim. Can be used to update the statusline.

FileReadCmd

FileReadCmd Before reading a file with a ":read" command. Should do the reading of the file. Cmd-event

FileReadPost

FileReadPost After reading a file with a ":read" command. Note that Vim sets the '[and '] marks to the first and last line of the read. This can be used to operate on the lines just read.

FileReadPre

FileReadPre Before reading a file with a ":read" command.

FileType

FileType When the 'filetype' option has been set. The pattern is matched against the filetype. <afile> is the name of the file where this option was set. <amatch> is the new value of 'filetype'.
Cannot switch windows or buffers.
See filetypes.

FileWriteCmd

FileWriteCmd Before writing to a file, when not writing the whole buffer. Should do the writing to the file. Should not change the buffer. Use the '[and '] marks for the range of lines.
Cmd-event

FileWritePost

FileWritePost After writing to a file, when not writing the whole buffer.

FileWritePre

FileWritePre Before writing to a file, when not writing the whole buffer. Use the '[and '] marks for the range of lines.

FilterReadPost

FilterReadPost After reading a file from a filter command. Vim checks the pattern against the name of the current buffer as with FilterReadPre.
Not triggered when 'shelltemp' is off.

FilterReadPre E135

FilterReadPre Before reading a file from a filter command. Vim checks the pattern against the name of the current buffer, not the name of the temporary file that is the output of the filter command.
Not triggered when 'shelltemp' is off.

FilterWritePost

FilterWritePost After writing a file for a filter command or making a diff with an external diff (see DiffUpdated for internal diff).
Vim checks the pattern against the name of the current buffer as with FilterWritePre.
Not triggered when 'shelltemp' is off.

FilterWritePre

FilterWritePre Before writing a file for a filter command or making a diff with an external diff.
Vim checks the pattern against the name of the current buffer, not the name of the temporary file that is the output of the filter command.
Not triggered when 'shelltemp' is off.

FocusGained

FocusGained Nvim got focus.

FocusLost

FocusLost Nvim lost focus. Also (potentially) when a GUI dialog pops up.

FuncUndefined

FuncUndefined
When a user function is used but it isn't defined. Useful for defining a function only when it's used. The pattern is matched against the function name. Both <amatch> and <afile> are set to the name of the function. This is triggered even when inside an autocommand defined without autocmd-nested. **NOTE:** When writing Vim scripts a better alternative is to use an autoloaded function. See autoload-functions.

InsertChange

InsertChange
When typing <Insert> while in Insert or Replace mode. The v:insertmode variable indicates the new mode.
Be careful not to move the cursor or do anything else that the user does not expect.

InsertCharPre

InsertCharPre
When a character is typed in Insert mode, before inserting the char.
The v:char variable indicates the char typed and can be changed during the event to insert a different character. When v:char is set to more than one character this text is inserted literally.

Not triggered for . redo, i_CTRL-R, i_CTRL-A.

Cannot change the text. textlock

InsertEnter

InsertEnter
Just before starting Insert mode. Also for Replace mode and Virtual Replace mode. The v:insertmode variable indicates the mode.
Be careful not to do anything else that the user does not expect.
The cursor is restored afterwards. If you do not want that set v:char to a non-empty string.

InsertLeavePre

InsertLeavePre
Just before leaving Insert mode. Also when using CTRL-O i_CTRL-O. Be careful not to change mode or use :normal, it will likely cause trouble.

InsertLeave

InsertLeave
Just after leaving Insert mode. Also when using CTRL-O i_CTRL-O. But not for i_CTRL-C.

LspAttach
See LspAttach
LspDetach
See LspDetach
LspNotify
See LspNotify
LspProgress
See LspProgress
LspRequest
See LspRequest
LspTokenUpdate
See LspTokenUpdate

MarkSet

MarkSet
After a mark is set by m, :mark, and nvim_buf_set_mark(), or deleted by :delmarks and nvim_buf_del_mark(). Supports [a-zA-Z] marks (may support more in the future).
The autocmd-pattern is matched against the mark name (e.g. [ab] matches a or b, * matches all). If line is 0, the mark has been deleted.

The event-data has these keys (type: vim.event.markset.data):

- name: Mark name (e.g. "a")
- line: Mark line.
- col: Mark column.

MenuPopup

MenuPopup
Just before showing the popup menu (under the right mouse button). Useful for adjusting the menu for what is under the cursor or mouse pointer.
The pattern is matched against one or two characters representing the mode:

n	Normal
v	Visual
o	Operator-pending

```
                                        i       Insert
                                        c       Command line
                                        tl      Terminal
```

ModeChanged

```
ModeChanged                     After changing the mode.  The pattern is
                                matched against 'old_mode:new_mode', for
                                example match against *:c* to simulate
                                CmdlineEnter.
                                The following values of v:event are set:
                                   old_mode     The mode before it changed.
                                   new_mode     The new mode as also returned
                                                by mode() called with a
                                                non-zero argument.
                                When ModeChanged is triggered, old_mode will
                                have the value of new_mode when the event was
                                last triggered.
                                This will be triggered on every minor mode
                                change.
                                Usage example to use relative line numbers
                                when entering visual mode:
```

```
:au ModeChanged [vV\x16]*:* let &l:rnu = mode() =~# '^[vV\x16]'
:au ModeChanged *:[vV\x16]* let &l:rnu = mode() =~# '^[vV\x16]'
:au WinEnter,WinLeave * let &l:rnu = mode() =~# '^[vV\x16]'
```

OptionSet

```
OptionSet                       After setting an option (except during
                                startup).  The autocmd-pattern is matched
                                against the long option name.  <amatch>
                                indicates what option has been set.

                                v:option_type indicates whether it's global
                                or local scoped.
                                v:option_command indicates what type of
                                set/let command was used (follow the tag to
                                see the table).
                                v:option_new indicates the newly set value.
                                v:option_oldlocal has the old local value.
                                v:option_oldglobal has the old global value.
                                v:option_old indicates the old option value.

                                v:option_oldlocal is only set when :set
                                or :setlocal or a modeline was used to set
                                the option.  Similarly v:option_oldglobal is
                                only set when :set or :setglobal was used.

                                This does not set <abuf>, you could use
                                bufnr().

                                Note that when setting a global-local option
                                with :set, then v:option_old is the old
                                global value.  However, for all options that
                                are not global-local it is the old local
                                value.

                                Usage example: Check for the existence of the
                                directory in the 'backupdir' and 'undodir'
                                options, create the directory if it doesn't
                                exist yet.

                                Note: Do not reset the same option during this
                                autocommand, that may break plugins.  You can
                                always use :noautocmd to prevent triggering
                                OptionSet.

                                Note: OptionSet is deferred when triggered by
                                'modified', even when set via :set.

                                Non-recursive: :set in the autocommand does
                                not trigger OptionSet again.

                                Not triggered on startup.

PackChangedPre                  See PackChangedPre
PackChanged                     See PackChanged
```

Progress

```
                                After a progress-message is created or updated via
                                nvim_echo. The pattern is matched against
                                source of the message.

                                The event-data has these keys (type: vim.event.progress.data):
```

- id: id of the message
- text: text of the message
- title: title of the progress message
- source: source of the progress message

- status: status of the progress message
- percent: how much progress has been made
- data: arbitrary data sent with the message

Usage example:

```
vim.api.nvim_create_autocmd('Progress', {
  pattern={'term'},
  callback = function(ev)
    print(string.format('event fired: %s', vim.inspect(ev)))
  end
})
local id = vim.api.nvim_echo({{'searching...'}}, true, {kind='progress', status='running', percent=10, title='term', source='search'})
vim.api.nvim_echo({{'searching'}}, true, {id = id, kind='progress', status='running', percent=50, title='term', source='search'})
vim.api.nvim_echo({{'done'}}, true, {id = id, kind='progress', status='success', percent=100, title='term', source='search'})
```

QuickFixCmdPre

QuickFixCmdPre — Before a quickfix command is run (:make, :lmake, :grep, :lgrep, :grepadd, :lgrepadd, :vimgrep, :lvimgrep, :vimgrepadd, :lvimgrepadd, :cfile, :cgetfile, :caddfile, :lfile, :lgetfile, :laddfile, :helpgrep, :lhelpgrep, :cexpr, :cgetexpr, :caddexpr, :cbuffer, :cgetbuffer, :caddbuffer). The pattern is matched against the command being run. When :grep is used but 'grepprg' is set to "internal" it still matches "grep". This command cannot be used to set the 'makeprg' and 'grepprg' variables. If this command causes an error, the quickfix command is not executed.

QuickFixCmdPost

QuickFixCmdPost — Like QuickFixCmdPre, but after a quickfix command is run, before jumping to the first location. For :cfile and :lfile commands it is run after the error file is read and before moving to the first error. See QuickFixCmdPost-example.

QuitPre

QuitPre — When using :quit, :wq or :qall, before deciding whether it closes the current window or quits Vim. For :wq the buffer is written before QuitPre is triggered. Can be used to close any non-essential window if the current window is the last ordinary window. See also ExitPre, WinClosed.

RecordingEnter

RecordingEnter — When a macro starts recording. The pattern is the current file name, and reg_recording() is the current register that is used.

RecordingLeave

RecordingLeave — When a macro stops recording. The pattern is the current file name, and reg_recording() is the recorded register. reg_recorded() is only updated after this event. Sets these v:event keys:
    regcontents
    regname

RemoteReply

RemoteReply — When a reply from a Vim that functions as server was received server2client(). The pattern is matched against the {serverid}. <amatch> is equal to the {serverid} from which the reply was sent, and <afile> is the actual reply string. Note that even if an autocommand is defined, the reply should be read with remote_read() to consume it.

SafeState

SafeState When nothing is pending, going to wait for the user to type a character. This will not be triggered when:

- an operator is pending
- a register was entered with "r
- halfway executing a command
- executing a mapping
- there is typeahead
- Insert mode completion is active
- Command line completion is active

You can use mode() to find out what state Vim is in. That may be:

- Visual mode
- Normal mode
- Insert mode
- Command-line mode

Depending on what you want to do, you may also check more with state(), e.g. whether the screen was scrolled for messages.

SearchWrapped

SearchWrapped After making a search with n or N if the search wraps around the document back to the start/finish respectively.

SessionLoadPre

SessionLoadPre Before loading the session file created using the :mksession command.

SessionLoadPost

SessionLoadPost After loading the session file created using the :mksession command.

SessionWritePre

SessionWritePre Before writing the session file created using the :mksession command.

SessionWritePost

SessionWritePost After writing a session file by calling the :mksession command.

ShellCmdPost

ShellCmdPost After executing a shell command with :!cmd, :make and :grep. Can be used to check for any changed files.

For non-blocking shell commands, see TermClose job-control.

ShellFilterPost

ShellFilterPost After executing a shell command with ":{range}!cmd", ":w !cmd" or ":r !cmd". Can be used to check for any changed files.

Signal

Signal After Nvim receives a signal. The pattern is matched against the signal name. Only "SIGUSR1" and "SIGWINCH" are supported. This is triggered even when inside an autocommand defined without autocmd-nested. Example:

```
autocmd Signal SIGUSR1 call some#func()
```

SourceCmd

SourceCmd When sourcing a Vimscript/Lua file. :source
<afile> is the name of the file being sourced.
The autocommand must source that file.
Cmd-event

SourcePre

SourcePre Before sourcing a Vimscript/Lua file. :source
<afile> is the name of the file being sourced.

SourcePost

SourcePost After sourcing a Vimscript/Lua file. :source
<afile> is the name of the file being sourced.
Not triggered when sourcing was interrupted.
Also triggered after a SourceCmd autocommand
was triggered.

SpellFileMissing

SpellFileMissing When trying to load a spell checking file and
it can't be found. The pattern is matched
against the language. <amatch> is the
language, 'encoding' also matters. See
spell-SpellFileMissing.

StdinReadPost

StdinReadPost During startup, after reading from stdin into
the buffer, before executing modelines. --

StdinReadPre

StdinReadPre During startup, before reading from stdin into
the buffer. --

SwapExists

SwapExists Detected an existing swap file when starting
to edit a file. Only when it is possible to
select a way to handle the situation, when Vim
would ask the user what to do.
The v:swapname variable holds the name of
the swap file found, <afile> the file being
edited. v:swapcommand may contain a command
to be executed in the opened file.
The commands should set the v:swapchoice
variable to a string with one character to
tell Vim what should be done next:

```
'o'     open read-only
'e'     edit the file anyway
'r'     recover
'd'     delete the swap file
'q'     quit, don't edit the file
'a'     abort, like hitting CTRL-C
```

When set to an empty string the user will be
asked, as if there was no SwapExists autocmd.

E812

Cannot change to another buffer, change
the buffer name or change directory.

Syntax

Syntax When the 'syntax' option has been set. The
pattern is matched against the syntax name.
<afile> expands to the name of the file where
this option was set. <amatch> expands to the
new value of 'syntax'.
See :syn-on.

TabClosed

TabClosed After closing a tabpage. <afile> expands to
the closing tabpage-number.

TabClosedPre

TabClosedPre Before closing a tabpage. The current tabpage
(tabpagenr()) is the one being closed. The
window layout is locked, thus opening and
closing of windows is prohibited.

TabEnter

TabEnter Just after entering a tabpage.
After WinEnter.
Before BufEnter.

TabLeave

TabLeave Just before leaving a tabpage.
After WinLeave.

TabMoved

TabMoved After a tabpage is moved (by :tabmove or mouse drag). <amatch> expands to the old tabpage-number before the move.

The event-data has these keys (type: vim.event.tabmoved.data)

- tabnr_old: tabpage-number before the move.
- tabnr_new: tabpage-number after the move.

TabNew

TabNew When creating a new tabpage.
After WinEnter.
Before TabEnter.

TabNewEntered

TabNewEntered After entering a new tabpage.
After BufEnter.

TermClose

TermClose When a terminal job ends.
Sets these v:event keys:
status job exit status, or -1 if the terminal buffer is deleted

TermEnter

TermEnter After entering Terminal-mode.
After TermOpen.

TermLeave

TermLeave After leaving Terminal-mode.
After TermClose.

TermOpen

TermOpen When a terminal job is starting. Can be used to configure the terminal buffer. To get the terminal process details:

```
vim.print(vim.api.nvim_get_chan_info(vim.bo.channel))
```

TermRequest

TermRequest When a :terminal child process emits an OSC, DCS, or APC sequence. Sets v:termrequest.

The event-data has these keys (type: vim.event.termrequest.data):

- sequence: the received sequence
- terminator: the received sequence terminator (i.e. BEL or ST)
- cursor: (1,0)-indexed, buffer-relative

position of the cursor when the sequence was received (line number may be <= 0 if the position is no longer in the buffer)

This is triggered even when inside an autocommand defined without autocmd-nested.

TermResponse

TermResponse When Nvim receives an OSC, DCS, APC, or recognized CSI (DA1, kitty multiple-cursors) response from the host terminal. Sets v:termresponse.

The event-data has these keys (type: vim.event.termresponse.data):

- chan: the source channel id
- sequence: the received sequence

This is triggered even when inside an autocommand defined without autocmd-nested.

May be triggered during another event (file I/O, a shell command, or anything else that takes time).

Example:

```
-- Query the terminal palette for the RGB value of color 1
-- (red) using OSC 4
vim.api.nvim_create_autocmd('TermResponse', {
  once = true,
  callback = function(ev)
    local resp = ev.data.sequence
    local r, g, b = resp:match("\027%]4;1;rgb:(%w+)/(%w+)/(%w+)")
  end,
})
vim.api.nvim_ui_send("\027]4;1;?\027\\")
```

TextChanged

TextChanged — After a change was made to the text in the current buffer in Normal mode. That is after b:changedtick has changed (also when that happened before the TextChanged autocommand was defined).
Not triggered when there is typeahead or when an operator is pending.
Note: Cannot be skipped with :noautocmd.
Careful: This is triggered very often, don't do anything that the user does not expect or that is slow.

TextChangedI

TextChangedI — After a change was made to the text in the current buffer in Insert mode.
Not triggered when the popup menu is visible.
Otherwise the same as TextChanged.

TextChangedP

TextChangedP — After a change was made to the text in the current buffer in Insert mode, only when the popup menu is visible. Otherwise the same as TextChanged.

TextChangedT

TextChangedT — After a change was made to the text in the current buffer in Terminal-mode. Otherwise the same as TextChanged.

TextPutPost

TextPutPost — After text has been put in the current buffer.
The following values in v:event are mostly the same as TextYankPost:

operator	The operation performed, either 'p' or 'P'.
regcontents	Text that was put. For quote_=, this is the result of the expression.
regname	Name of the register or empty string for the unnamed register. For nvim_put() this is set to '_'.
regtype	Type of the register, see getregtype().
visual	True if the operation is performed in Visual mode.

Not triggered when quote_ is used nor when called recursively.
It is not allowed to change the buffer text, see textlock.
Note that for the quote_. register, since the last inserted text is buffered into the input buffer (buffer isn't modified directly), this autocommand is called directly after TextPutPre.

TextPutPre

TextPutPre — Before text has been put in the current buffer.
Same values as TextPutPost in v:event. It is valid to call setreg() in this autocommand, allowing you to process and modify the text in "regcontents" before it is put. However this does not apply to quote_#, quote_=, quote_%, quote_:, quote_/ or quote_.
Not triggered when quote_ is used nor when called recursively.
It is not allowed to change the buffer text, see textlock.

TextYankPost

TextYankPost — Just after a yank or deleting command, but not if the black hole register quote_ is used nor for setreg(). Pattern must be "*".

Sets these v:event keys:
    inclusive
    operator
    regcontents
    regname
    regtype
    visual
The inclusive flag combined with the '[
and '] marks can be used to calculate the
precise region of the operation.

Non-recursive (event cannot trigger itself).
Cannot change the text. textlock

UIEnter

UIEnter After a UI connects via nvim_ui_attach(), or
after builtin TUI is started, after VimEnter.
Sets these v:event keys:
    chan: channel-id of the UI

UILeave

UILeave After a UI disconnects from Nvim, or after
builtin TUI is stopped, after VimLeave.
Sets these v:event keys:
    chan: channel-id of the UI

User

User Not executed automatically. Use :doautocmd
to trigger this, typically for "custom events"
in a plugin. Example:

```
:autocmd User MyPlugin echom 'got MyPlugin event'
:doautocmd User MyPlugin
```

UserGettingBored

UserGettingBored When the user presses the same key 42 times.
Just kidding! :-)

VimEnter

VimEnter After doing all the startup stuff, including
loading vimrc files, executing the "-c cmd"
arguments, creating all windows and loading
the buffers in them.
Just before this event is triggered the
v:vim_did_enter variable is set, so that you
can do:

```
if v:vim_did_enter
  call s:init()
else
  au VimEnter * call s:init()
endif
```

VimLeave

VimLeave Before exiting Vim, just after writing the
.shada file. Executed only once, like
VimLeavePre.
Use v:dying to detect an abnormal exit.
Use v:exiting to get the exit code.
Not triggered if v:dying is 2 or more.

VimLeavePre

VimLeavePre Before exiting Vim, just before writing the
shada file. Executed only once, if the
pattern matches the current buffer on exit.
Mostly useful with a "*" pattern.

```
:autocmd VimLeavePre * call CleanupStuff()
```

Use v:dying to detect an abnormal exit.
Use v:exiting to get the exit code.
Not triggered if v:dying is 2 or more.

VimResized

VimResized After Nvim itself was resized: 'lines' and/or
'columns' changed. But not during startup.

To ensure the window layout is always
"equalized" on resize:

```
:autocmd VimResized * wincmd =
```

VimResume

VimResume After Nvim resumes from suspend state.

VimSuspend

VimSuspend — Before Nvim enters suspend state.

WinClosed

WinClosed — When closing a window, just before it is removed from the window layout. The pattern is matched against the window-ID. Both <amatch> and <afile> are set to the window-ID. After WinLeave.
Non-recursive (event cannot trigger itself).
See also ExitPre, QuitPre.

WinEnter

WinEnter — After entering another window. Not done for the first window, when Vim has just started. Useful for setting the window height.
If the window is for another buffer, Vim executes the BufEnter autocommands after the WinEnter autocommands.
Note: For split and tabpage commands the WinEnter event is triggered after the split or tab command but before the file is loaded.

WinLeave

WinLeave — Before leaving a window. If the window to be entered next is for a different buffer, Vim executes the BufLeave autocommands before the WinLeave autocommands (but not for ":new").
Not used for ":qa" or ":q" when exiting Vim.
Before WinClosed.

WinNewPre

WinNewPre — Before creating a new window. Triggered before commands that modify window layout by creating a split.
Not done when creating tabpages and for the first window, as the window structure is not initialized yet and so is generally not safe.
It is not allowed to modify window layout while executing commands for the WinNewPre event.
Most useful to store current window layout and compare it with the new layout after the Window has been created.

WinNew

WinNew — When a new window was created. Not done for the first window, when Vim has just started.
Before WinEnter.

WinResized

WinResized — After a window in the current tabpage changed width or height.
See win-scrolled-resized.

v:event is set with information about size changes. WinResized-event

Same behavior as WinScrolled for the pattern, triggering and recursiveness.

WinScrolled

WinScrolled — After any window in the current tabpage scrolled the text (horizontally or vertically) or changed width or height. See win-scrolled-resized.

Note: This can not be skipped with :noautocmd, because it triggers after processing normal commands when Vim is back in the main loop. If you want to disable this, consider setting the 'eventignore' option instead.

The pattern is matched against the window-ID of the first window that scrolled or resized. Both <amatch> and <afile> are set to the window-ID.

v:event is set with information about size and scroll changes. WinScrolled-event

Only starts triggering after startup finished and the first screen redraw was done.
Does not trigger when defining the first WinScrolled or WinResized event, but may trigger when adding more.

Non-recursive: the event will not trigger while executing commands for the WinScrolled

event. However, if the command causes a
window to scroll or change size, then another
WinScrolled event will be triggered later.

6. Patterns

autocmd-pattern {aupat}

The {aupat} argument of :autocmd can be a comma-separated list. This works as
if the command was given with each pattern separately. Thus this command:

```
:autocmd BufRead *.txt,*.info set et
```

Is equivalent to:

```
:autocmd BufRead *.txt set et
:autocmd BufRead *.info set et
```

The file pattern {aupat} is tested for a match against the file name in one of
two ways:
 1. When there is no '/' in the pattern, Vim checks for a match against only

     the tail part of the file name (without its leading directory path).
 2. When there is a '/' in the pattern, Vim checks for a match against both the

     short file name (as you typed it) and the full file name (after expanding
     it to a full path and resolving symbolic links).

The special pattern <buffer> or <buffer=N> is used for buffer-local
autocommands autocmd-buflocal. This pattern is not matched against the name
of a buffer.

Examples:

```
:autocmd BufRead *.txt          set et
```

Set the 'et' option for all text files.

```
:autocmd BufRead /vim/src/*.c   set cindent
```

Set the 'cindent' option for C files in the /vim/src directory.

```
:autocmd BufRead /tmp/*.c       set ts=5
```

If you have a link from "/tmp/test.c" to "/home/nobody/vim/src/test.c", and
you start editing "/tmp/test.c", this autocommand will match.

Note: To match part of a path, but not from the root directory, use a "*" as
the first character. Example:

```
:autocmd BufRead */doc/*.txt    set tw=78
```

This autocommand will for example be executed for "/tmp/doc/xx.txt" and
"/usr/home/piet/doc/yy.txt". The number of directories does not matter here.

The file name that the pattern is matched against is after expanding
wildcards. Thus if you issue this command:

```
:e $ROOTDIR/main.$EXT
```

The argument is first expanded to:

```
/usr/root/main.py
```

Before it's matched with the pattern of the autocommand. Careful with this
when using events like FileReadCmd, the value of <amatch> may not be what you
expect.

Environment variables can be used in a pattern:

```
:autocmd BufRead $VIMRUNTIME/doc/*.txt  set expandtab
```

And ~ can be used for the home directory (if $HOME is defined):

```
:autocmd BufWritePost ~/.config/nvim/init.vim   so <afile>
:autocmd BufRead ~archive/*      set readonly
```

The environment variable is expanded when the autocommand is defined, not when
the autocommand is executed. This is different from the command!

file-pattern

The pattern is interpreted like mostly used in file names:

```
        *       matches any sequence of characters; Unusual: includes path
                separators
        ?       matches any single character
        \?      matches a '?'
        .       matches a '.'
        ~       matches a '~'
        ,       separates patterns
        \,      matches a ','
        { }     like \( \) in a pattern
        ,       inside { }: like \| in a pattern
        \}      literal }
        \{      literal {
        \\\{n,m\}  like \{n,m} in a pattern
        \       special meaning like in a pattern
        [ch]    matches 'c' or 'h'
        [^ch]   match any character but 'c' and 'h'
```

Note that for all systems the '/' character is used for path separator (even for MS-Windows). This was done because the backslash is difficult to use in a pattern and to make the autocommands portable across different systems. To only match a '/' on all platforms (e.g. in a non-file pattern), use "\/".

It is possible to use pattern items, but they may not work as expected, because of the translation done for the above.

autocmd-changes

Matching with the pattern is done when an event is triggered. Changing the buffer name in one of the autocommands, or even deleting the buffer, does not change which autocommands will be executed. Example:

```
au BufEnter *.foo  bdel
au BufEnter *.foo  set modified
```

This will delete the current buffer and then set 'modified' in what has become the current buffer instead. Vim doesn't take into account that "*.foo" doesn't match with that buffer name. It matches "*.foo" with the name of the buffer at the moment the event was triggered.

However, buffer-local autocommands will not be executed for a buffer that has been wiped out with :bwipe. After deleting the buffer with :bdel the buffer actually still exists (it becomes unlisted), thus the autocommands are still executed.

7. Buffer-local autocommands autocmd-buflocal autocmd-buffer-local

<buffer> <buffer=N> <buffer=abuf> E680

Buffer-local autocommands are attached to a specific buffer. They are useful if the buffer does not have a name and when the name does not match a specific pattern. But it also means they must be explicitly added to each buffer.

Instead of a pattern buffer-local autocommands use one of these forms:

```
        <buffer>        current buffer
        <buffer=99>     buffer number 99
        <buffer=abuf>   using <abuf> (only when executing autocommands)
                        <abuf>
```

Examples:

```
:au CursorHold <buffer>  echo 'hold'
:au CursorHold <buffer=33>  echo 'hold'
:au BufNewFile * au CursorHold <buffer=abuf>  echo 'hold'
```

All the commands for autocommands also work with buffer-local autocommands, simply use the special string instead of the pattern. Examples:

```
:au! * <buffer>             " remove buffer-local autocommands for
                                " current buffer
:au! * <buffer=33>                  " remove buffer-local autocommands for
                                " buffer #33
:bufdo :au! CursorHold <buffer>  " remove autocmd for given event for all
                                " buffers
:au * <buffer>              " list buffer-local autocommands for
                                " current buffer
```

Note that when an autocommand is defined for the current buffer, it is stored with the buffer number. Thus it uses the form "<buffer=12>", where 12 is the number of the current buffer. You will see this when listing autocommands, for example.

To test for presence of buffer-local autocommands use the exists() function as follows:

```
:if exists("#CursorHold#<buffer=12>") | ... | endif
:if exists("#CursorHold#<buffer>") | ... | endif    " for current buffer
```

When a buffer is wiped out its buffer-local autocommands are also gone, of course. Note that when deleting a buffer, e.g., with ":bdel", it is only

unlisted, the autocommands are still present. In order to see the removal of buffer-local autocommands:

```
:set verbose=6
```

It is not possible to define buffer-local autocommands for a non-existent buffer.

8. Groups

autocmd-groups

Autocommands can be put together in a group. This is useful for removing or executing a group of autocommands. For example, all the autocommands for syntax highlighting are put in the "highlight" group, to be able to execute ":doautoall highlight BufRead" when the GUI starts.

When no specific group is selected, Vim uses the default group. The default group does not have a name. You cannot execute the autocommands from the default group separately; you can execute them only by executing autocommands for all groups.

Normally, when executing autocommands automatically, Vim uses the autocommands for all groups. The group only matters when executing autocommands with ":doautocmd" or ":doautoall", or when defining or deleting autocommands.

The group name can contain any characters except white space. The group name "end" is reserved (also in uppercase).

The group name is case sensitive. Note that this is different from the event name!

:aug :augroup

:aug[roup] {name} Define the autocmd group name for the following ":autocmd" commands. The name "end" or "END" selects the default group.
To avoid confusion, the name should be different from existing {event} names, as this most likely will not do what you intended.

:augroup-delete E367 W19 E936

:aug[roup]! {name} Delete the autocmd group {name}. Don't use this if there is still an autocommand using this group! You will get a warning if doing it anyway. When the group is the current group you will get error E936.

To enter autocommands for a specific group, use this method:

1. Select the group with ":augroup {name}".
2. Delete any old autocommands with ":au!".
3. Define the autocommands.
4. Go back to the default group with "augroup END".

Example:

```
:augroup uncompress
:  au!
:  au BufEnter *.gz     %!gunzip
:augroup END
```

This prevents having the autocommands defined twice (e.g., after sourcing the vimrc file again).

FileExplorer

There is one group that is recognized by Vim: FileExplorer. If this group exists Vim assumes that editing a directory is possible and will trigger a plugin that lists the files in that directory. This is used by directory browser plugins. This allows you to do:

```
browse edit
```

9. Executing autocommands

autocmd-execute

Vim can also execute Autocommands non-automatically. This is useful if you have changed autocommands, or when Vim has executed the wrong autocommands (e.g., the file pattern match was wrong).

Note that the 'eventignore' option applies here too. Events listed in this option will not cause any commands to be executed.

:do :doau :doaut :doautocmd E217

:do[autocmd] [<nomodeline>] [group] {event} [fname]
Apply the autocommands matching [fname] (default: current file name) for {event} to the current buffer.

You can use this when the current file name does not
match the right pattern, after changing settings, or
to execute autocommands for a certain event.
It's possible to use this inside an autocommand too,
so you can base the autocommands for one extension on
another extension. Example:

```
:au BufEnter *.cpp so ~/.config/nvim/init_cpp.vim
:au BufEnter *.cpp doau BufEnter x.c
```

Be careful to avoid endless loops. autocmd-nested

When the [group] argument is not given, Vim executes
the autocommands for all groups. When the [group]
argument is included, Vim executes only the matching
autocommands for that group. Undefined group is an
error.

<nomodeline>

After applying the autocommands the modelines are
processed, so that their settings overrule the
settings from autocommands when editing a file. This
is skipped if <nomodeline> is specified. You probably
want to use <nomodeline> for events not used when
loading a buffer, such as User.
Modelines are also skipped when no matching
autocommands were executed.

:doautoa :doautoall

:doautoa[ll] [<nomodeline>] [group] {event} [fname]
Like ":doautocmd", but apply the autocommands to each
loaded buffer. The current buffer is done last.

Note that [fname] is used to select the autocommands,
not the buffers to which they are applied. Example:

```
augroup mine
  autocmd!
  autocmd FileType * echo expand('<amatch>')
augroup END
doautoall mine FileType Loaded-Buffer
```

Sourcing this script, you'll see as many
"Loaded-Buffer" echoed as there are loaded buffers.

Careful: Don't use this for autocommands that delete a
buffer, change to another buffer or change the
contents of a buffer; the result is unpredictable.
This command is intended for autocommands that set
options, change highlighting, and things like that.

10. Using autocommands

autocmd-use

For WRITING FILES there are four possible sets of events. Vim uses only one
of these sets for a write command:

BufWriteCmd	BufWritePre	BufWritePost	writing the whole buffer
	FilterWritePre	FilterWritePost	writing to filter temp file
FileAppendCmd	FileAppendPre	FileAppendPost	appending to a file
FileWriteCmd	FileWritePre	FileWritePost	any other file write

When there is a matching "*Cmd" autocommand, it is assumed it will do the
writing. No further writing is done and the other events are not triggered.
Cmd-event

Note that the "*WritePost" commands should undo any changes to the buffer that
were caused by the "*WritePre" commands; otherwise, writing the file will have
the side effect of changing the buffer.

Before executing the autocommands, the buffer from which the lines are to be
written temporarily becomes the current buffer. Unless the autocommands
change the current buffer or delete the previously current buffer, the
previously current buffer is made the current buffer again.

The "*WritePre" and "*AppendPre" autocommands must not delete the buffer from
which the lines are to be written.

The '[and '] marks have a special position:

- Before the "*ReadPre" event the '[mark is set to the line just above where
  the new lines will be inserted.
- Before the "*ReadPost" event the '[mark is set to the first line that was
  just read, the '] mark to the last line.
- Before executing the "*WriteCmd", "*WritePre" and "*AppendPre" autocommands the '[
  mark is set to the first line that will be written, the '] mark to the last
  line.

  Careful: '[and '] change when using commands that change the buffer.

In commands which expect a file name, you can use "<afile>" for the file name
that is being read :<afile> (you can also use "%" for the current file
name). "<abuf>" can be used for the buffer number of the currently effective

buffer. This also works for buffers that don't have a name. But it doesn't
work for files without a buffer (e.g., with ":r file").

gzip-example

Examples for reading and writing compressed files:

```
:augroup gzip
:  autocmd!
:  autocmd BufReadPre,FileReadPre       *.gz set bin
:  autocmd BufReadPost,FileReadPost     *.gz '[,']!gunzip
:  autocmd BufReadPost,FileReadPost     *.gz set nobin
:  autocmd BufReadPost,FileReadPost     *.gz execute ":doautocmd BufReadPost " .. expand("%:r")
:  autocmd BufWritePost,FileWritePost   *.gz !mv <afile> <afile>:r
:  autocmd BufWritePost,FileWritePost   *.gz !gzip <afile>:r

:  autocmd FileAppendPre                *.gz !gunzip <afile>
:  autocmd FileAppendPre                *.gz !mv <afile>:r <afile>
:  autocmd FileAppendPost               *.gz !mv <afile> <afile>:r
:  autocmd FileAppendPost               *.gz !gzip <afile>:r
:augroup END
```

The "gzip" group is used to be able to delete any existing autocommands with
":autocmd!", for when the file is sourced twice.

("<afile>:r" is the file name without the extension, see :_%:)

The commands executed for the BufNewFile, BufRead/BufReadPost, BufWritePost,
FileAppendPost and VimLeave events do not set or reset the changed flag of the
buffer. When you decompress the buffer with the BufReadPost autocommands, you
can still exit with ":q". When you use ":undo" in BufWritePost to undo the
changes made by BufWritePre commands, you can still do ":q" (this also makes
"ZZ" work). If you do want the buffer to be marked as modified, set the
'modified' option.

To execute Normal mode commands from an autocommand, use the ":normal"
command. Use with care! If the Normal mode command is not finished, the user
needs to type characters (e.g., after ":normal m" you need to type a mark
name).

If you want the buffer to be unmodified after changing it, reset the
'modified' option. This makes it possible to exit the buffer with ":q"
instead of ":q!".

autocmd-nested E218

By default, autocommands do not nest. For example, if you use ":e" or ":w" in
an autocommand, Vim does not execute the BufRead and BufWrite autocommands for
those commands. If you do want this, use the "++nested" flag for those
commands in which you want nesting. For example:

```
:autocmd FileChangedShell *.c ++nested e!
```

The nesting is limited to 10 levels to get out of recursive loops.

It's possible to use the ":au" command in an autocommand. This can be a
self-modifying command! This can be useful for an autocommand that should
execute only once.

If you want to skip autocommands for one command, use the :noautocmd command
modifier or the 'eventignore' option.

Note: When reading a file (with ":read file" or with a filter command) and the
last line in the file does not have an <EOL>, Vim remembers this. At the next
write (with ":write file" or with a filter command), if the same line is
written again as the last line in a file AND 'binary' is set, Vim does not
supply an <EOL>. This makes a filter command on the just read lines write the
same file as was read, and makes a write command on just filtered lines write
the same file as was read from the filter. For example, another way to write
a compressed file:

```
:autocmd FileWritePre *.gz   set bin|'[,']!gzip
:autocmd FileWritePost *.gz  undo|set nobin
```

autocommand-pattern

You can specify multiple patterns, separated by commas. Here are some
examples:

```
:autocmd BufRead   *            set tw=79 nocin ic infercase fo=2croq
:autocmd BufRead   .letter      set tw=72 fo=2tcrq
:autocmd BufEnter  .letter      set dict=/usr/lib/dict/words
:autocmd BufLeave  .letter      set dict=
:autocmd BufRead,BufNewFile   *.c,*.h   set tw=0 cin noic
:autocmd BufEnter  *.c,*.h      abbr FOR for (i = 0; i < 3; ++i)<CR>{<CR>}<Esc>O
:autocmd BufLeave  *.c,*.h      unabbr FOR
```

For makefiles (makefile, Makefile, imakefile, makefile.unix, etc.):

```
:autocmd BufEnter  ?akefile*    set include=^s\=include
:autocmd BufLeave  ?akefile*    set include&
```

To always start editing C files at the first function:

```
:autocmd BufRead   *.c,*.h      1;/^{
```

Without the "1;" above, the search would start from wherever the file was
entered, rather than from the start of the file.

skeleton template

To read a skeleton (template) file when opening a new file:

```
:autocmd BufNewFile  *.c        0r ~/vim/skeleton.c
:autocmd BufNewFile  *.h        0r ~/vim/skeleton.h
:autocmd BufNewFile  *.java     0r ~/vim/skeleton.java
```

To insert the current date and time in a "*.html" file when writing it:

```
:autocmd BufWritePre,FileWritePre *.html   ks|call LastMod()|'s
:fun LastMod()
:  if line("$") > 20
:    let l = 20
:  else
:    let l = line("$")
:  endif
:  exe "1," .. l .. "g/Last modified: /s/Last modified: .*/Last modified: " ..
:  \ strftime("%Y %b %d")
:endfun
```

You need to have a line "Last modified: <date time>" in the first 20 lines
of the file for this to work. Vim replaces <date time> (and anything in the
same line after it) with the current date and time. Explanation:
        ks   mark current position with mark 's'
        call LastMod() call the LastMod() function to do the work
        's   return the cursor to the old position
The LastMod() function checks if the file is shorter than 20 lines, and then
uses the ":g" command to find lines that contain "Last modified: ". For those
lines the ":s" command is executed to replace the existing date with the
current one. The ":execute" command is used to be able to use an expression
for the ":g" and ":s" commands. The date is obtained with the strftime()
function. You can change its argument to get another date string.

When entering :autocmd on the command-line, completion of events and command
names may be done (with <Tab>, CTRL-D, etc.) where appropriate.

Vim executes all matching autocommands in the order that you specify them.
It is recommended that your first autocommand be used for all files by using
"*" as the file pattern. This means that you can define defaults you like
here for any settings, and if there is another matching autocommand it will
override these. But if there is no other matching autocommand, then at least
your default settings are recovered (if entering this file from another for
which autocommands did match). Note that "*" will also match files starting
with ".", unlike Unix shells.

autocmd-searchpat

Autocommands do not change the current search patterns. Vim saves the current
search patterns before executing autocommands then restores them after the
autocommands finish. This means that autocommands do not affect the strings
highlighted with the 'hlsearch' option. Within autocommands, you can still
use search patterns normally, e.g., with the "n" command.
If you want an autocommand to set the search pattern, such that it is used
after the autocommand finishes, use the ":let @/ =" command.
The search-highlighting cannot be switched off with ":nohlsearch" in an
autocommand. Use the 'h' flag in the 'shada' option to disable search-
highlighting when starting Vim.

Cmd-event

When using one of the "*Cmd" events, the matching autocommands are expected to
do the file reading, writing or sourcing. This can be used when working with
a special kind of file, for example on a remote system.
CAREFUL: If you use these events in a wrong way, it may have the effect of
making it impossible to read or write the matching files! Make sure you test
your autocommands properly. Best is to use a pattern that will never match a
normal file name, for example "ftp://*".

When defining a BufReadCmd it will be difficult for Vim to recover a crashed
editing session. When recovering from the original file, Vim reads only those
parts of a file that are not found in the swap file. Since that is not
possible with a BufReadCmd, use the :preserve command to make sure the
original file isn't needed for recovery. You might want to do this only when
you expect the file to be modified.

For file read and write commands the v:cmdarg variable holds the "++enc="
and "++ff=" argument that are effective. These should be used for the command
that reads/writes the file. The v:cmdbang variable is one when "!" was
used, zero otherwise.

See $VIMRUNTIME/plugin/net.lua for examples.

11. Disabling autocommands

autocmd-disable

To disable autocommands for some time use the 'eventignore' option. Note that this may cause unexpected behavior, make sure you restore 'eventignore' afterwards, using a :try block with :finally.

To disable autocmds indefinitely in a specific window use the 'eventignorewin' option. This can only be used to ignore window and buffer related events.

:noautocmd :noa

To disable autocommands for just one command use the ":noautocmd" command modifier. This will set 'eventignore' to "all" for the duration of the following command. Example:

```
:noautocmd w fname.gz
```

This will write the file without triggering the autocommands defined by the gzip plugin.

Note that some autocommands are not triggered right away, but only later. This specifically applies to CursorMoved and TextChanged.

Nvim asynchronous IO

1. Introduction channel-intro

Channels are Nvim's way of communicating with external processes.

There are several ways to open a channel:

1. Through stdin/stdout when `nvim` is started with `--headless` and a startup script or `--cmd` command opens the stdio channel using stdioopen().
2. Through stdin, stdout and stderr of a process spawned by jobstart().
3. Through the PTY master end opened with `jobstart(…, {'pty': v:true})`.
4. By connecting to a TCP/IP socket or named pipe with sockconnect().
5. By another process connecting to a socket listened to by Nvim. This only supports RPC channels, see rpc-connecting.

Channels support multiple modes or protocols. In the most basic mode of operation, raw bytes are read and written to the channel. The RPC protocol, based on the msgpack-rpc standard, enables nvim and the process at the other end to send remote calls and events to each other. The builtin terminal-emulator is also implemented on top of PTY channels.

Channel Id channel-id

Each channel is identified by an integer id, unique for the life of the current Nvim session. Functions like stdioopen() return channel ids; functions like chansend() consume channel ids.

2. Reading and writing raw bytes channel-bytes

Channels opened by Vimscript functions operate with raw bytes by default. For a job channel using RPC, bytes can still be read over its stderr. Similarly, only bytes can be written to Nvim's own stderr.

channel-callback E904 E906 E921 E5407

- on_stdout({chan-id}, {data}, {name}) on_stdout
- on_stderr({chan-id}, {data}, {name}) on_stderr
- on_stdin({chan-id}, {data}, {name}) on_stdin
- on_data({chan-id}, {data}, {name}) on_data

Scripts can react to channel activity (received data) via callback functions assigned to the `on_stdout`, `on_stderr`, `on_stdin`, or `on_data` option keys. Callbacks should be fast: avoid potentially slow/expensive work.

Parameters:

- {chan-id} Channel handle. channel-id
- {data} Raw data (readfile()-style list of strings) read from the channel. EOF is a single-item list: `['']`. First and last items may be partial lines! channel-lines
- {name} Stream name (string) like "stdout", so the same function can handle multiple streams. Event names depend on how the channel was opened and in what mode/protocol.

channel-buffered

The callback is invoked immediately as data is available, where a single-item list `['']` indicates EOF (stream closed). Alternatively set the `stdout_buffered`, `stderr_buffered`, `stdin_buffered`, or `data_buffered` option keys to invoke the callback only after all output was gathered and the stream was closed. E5210
If a buffering mode is used without a callback, the data is saved in the stream {name} key of the options dict. It is an error if the key exists.

channel-lines

Stream event handlers receive data as it becomes available from the OS, thus the first and last items in the {data} list may be partial lines. Empty string completes the previous partial line. Examples (not including the final `['']` emitted at EOF):

- `foobar` may arrive as `['fo'], ['obar']`
- `foo\nbar` may arrive as
- `['foo','bar']`
- or `['foo',''], ['bar']`
- or `['foo'], ['','bar']`
- or `['fo'], ['o','bar']`

There are two ways to deal with this:

- 1. To wait for the entire output, use channel-buffered mode.
- 2. To read line-by-line, use the following code:

```
let s:lines = ['']
func! s:on_event(job_id, data, event) dict
  let eof = (a:data == [''])
  " Complete the previous line.
  let s:lines[-1] .= a:data[0]
  " Append (last item may be a partial line, until EOF).
  call extend(s:lines, a:data[1:])
endf
```

If the callback functions are Dictionary-functions, self refers to the options dictionary containing the callbacks. Partials can also be used as callbacks.

Data can be sent to the channel using the chansend() function. Here is a simple example, echoing some data through a cat-process:

```
function! s:OnEvent(id, data, event) dict
  let str = join(a:data, "\n")
  echomsg str
endfunction
let id = jobstart(['cat'], {'on_stdout': function('s:OnEvent') } )
call chansend(id, "hello!")
```

Here is an example of setting a buffer to the result of grep, but only after all data has been processed:

```
function! s:OnEvent(id, data, event) dict
  call nvim_buf_set_lines(2, 0, -1, v:true, a:data)
endfunction
let id = jobstart(['grep', '^[0-9]'], { 'on_stdout': function('s:OnEvent'),
                                      \ 'stdout_buffered':v:true } )

call chansend(id, "stuff\n10 PRINT \"NVIM\"\nxx")
" no output is received, buffer is empty

call chansend(id, "xx\n20 GOTO 10\nzz\n")
call chanclose(id, 'stdin')
" now buffer has result
```

For additional examples with jobs, see job-control.

channel-pty

Special case: PTY channels opened with `jobstart(..., {'pty': v:true})` do not preprocess ANSI escape sequences, these will be sent raw to the callback. However, change of PTY size can be signaled to the slave using jobresize(). See also terminal-emulator.

Terminal characteristics (termios) for :terminal and PTY channels are copied from the host TTY, or if Nvim is --headless it uses default values:

```
:echo system('nvim --headless +"te stty -a" +"sleep 1" +"1,/^$/print" +q')
```

3. Communicating with msgpack RPC channel-rpc

When channels are opened with the `rpc` option set to true, the channel can be used for remote method calls in both directions, see msgpack-rpc. Note that rpc channels are implicitly trusted and the process at the other end can invoke any API function!

4. Standard IO channel channel-stdio

Nvim uses stdin/stdout to interact with the user over the terminal interface (TUI). If Nvim is --headless the TUI is not started and stdin/stdout can be used as a channel. See also --embed.

Call stdioopen() during startup to open the stdio channel as channel-id 1. Nvim's stderr is always available as v:stderr, a write-only bytes channel.

Example:

```
func! OnEvent(id, data, event)
  if a:data == [""]
    quit
  end
  call chansend(a:id, map(a:data, {i,v -> toupper(v)}))
endfunc
call stdioopen({'on_stdin': 'OnEvent'})
```

Put this in `uppercase.vim` and run:

```
nvim --headless --cmd "source uppercase.vim"
```

5. Using a prompt buffer repl prompt-buffer

Prompt buffers provide a "prompt" interface: they are like regular buffers, except only the last section of the buffer is editable, and the user can "submit" the prompt by hitting Enter. The 'scrollback' option can be used to limit the number of history lines kept above the prompt. Useful for implementing:

- chat UI
- REPL or shell plugins
- advanced "picker" plugins

A prompt buffer is created by setting 'buftype' to "prompt". You would normally only do that in a new, empty buffer:

```
:set buftype=prompt
```

Prompt buffers do not implicitly set 'modified' on user input or other changes. But a user or plugin can set 'modified' explicitly, to prevent the buffer from being easily closed.

The user can edit and input text at the end of the buffer. Pressing Enter in the input section invokes the prompt_setcallback() callback, which is typically expected to process the prompt and show results by appending to the buffer. To input multiline text, use Shift-<Enter> to add a new line without submitting the prompt, or just put or paste multiline text.

Only the section starting with the mark ': of the buffer (after the prompt) is editable. The rest of the buffer is not modifiable with Normal mode commands, though it can be modified by functions such as append(). Using other commands may mess up the buffer.

After setting `buftype=prompt`:
- Nvim unsets the 'comments' option.
- Nvim does not automatically start Insert mode (use `:startinsert` if you want to enter Insert mode)

The prompt prefix defaults to "% ", but can be set with prompt_setprompt(). You can get the effective prompt prefix for with prompt_getprompt().

The user can go to Normal mode and navigate through the buffer. This can be useful to see older output or copy text.

By default during prompt insert-mode, the CTRL-W key can be used to start a window command, such as CTRL-W w to switch to the next window. (Use Shift-CTRL-W to delete a word). When leaving the window Insert mode will be stopped. When coming back to the prompt window Insert mode will be restored.

Any command that starts Insert mode, such as "a", "i", "A" and "I", will move the cursor to the last line. "A" will move to the end of the line, "I" to the start of the line.

emacs-eshell

Example: The following code snippet implements a "shell buffer" similar to Emacs "eshell". Because the interface is a prompt-buffer, you can edit the prompt using normal Nvim commands, unlike a :terminal shell. Each prompt is sent to the shell process started by jobstart(), and appended to the buffer when output is received:

```
local shell_job

-- Handles a line of user input.
local function on_submit(text)
  -- Send the text to the shell with Enter appended.
  vim.api.nvim_chan_send(shell_job, text .. "\n")
end

-- Handles output from the shell.
local function on_output(_, msg, _)
  -- Add shell output above the prompt.
  local buf = vim.api.nvim_get_current_buf()
  vim.fn.prompt_appendbuf(buf, msg)
end

-- Handles the shell exit.
local function on_exit(_, _, _)
  vim.cmd("quit!")
end

-- Start a shell in the background.
shell_job = vim.fn.jobstart({ "/bin/sh" }, {
  on_stdout = on_output,
  on_stderr = on_output,
  on_exit = on_exit,
})

vim.cmd("new")
vim.bo.buftype = "prompt"
local buf = vim.api.nvim_get_current_buf()
vim.fn.prompt_setcallback(buf, on_submit)
vim.fn.prompt_setprompt(buf, "shell command: ")

-- Start accepting shell commands.
vim.cmd("startinsert")
```

Command-line-mode Cmdline
Command-line mode cmdline Command-line mode-cmdline :

Command-line mode is used to enter Ex commands (":"), search patterns
("/" and "?"), and filter commands ("!").

Basic command line editing is explained in chapter 20 of the user manual
usr_20.txt.

1. Command-line editing cmdline-editing

Normally characters are inserted in front of the cursor position. You can
move around in the command-line with the left and right cursor keys. With the
<Insert> key, you can toggle between inserting and overstriking characters.

Note that if your keyboard does not have working cursor keys or any of the
other special keys, you can use ":cnoremap" to define another key for them.
For example, to define tcsh style editing keys: tcsh-style

```
:cnoremap <C-A> <Home>
:cnoremap <C-F> <Right>
:cnoremap <C-B> <Left>
:cnoremap <Esc>b <S-Left>
:cnoremap <Esc>f <S-Right>
```

(<> notation <>; type all this literally)

cmdline-too-long

When the command line is getting longer than what fits on the screen, only the
part that fits will be shown. The cursor can only move in this visible part,
thus you cannot edit beyond that.

cmdline-history history
The command-lines that you enter are remembered in a history table. You can
recall them with the up and down cursor keys. There are actually five
history tables:

- one for ':' commands
- one for search strings
- one for expressions
- one for input lines, typed for the input() function.
- one for debug mode commands

These are completely separate. Each history can only be accessed when
entering the same type of line.
Use the 'history' option to set the number of lines that are remembered.

Notes:

- When you enter a command-line that is exactly the same as an older one, the
  old one is removed (to avoid repeated commands moving older commands out of
  the history).
- Only commands that are typed are remembered. A command executed completely
  from a mapping is not put in the history.
- All searches are put in the search history, including the ones that come
  from commands like "*" and "#". But for a mapping, only the last search is
  remembered (to avoid that long mappings trash the history).

There is an automatic completion of names on the command-line; see
cmdline-completion.

c_CTRL-V

CTRL-V Insert next non-digit literally. Up to three digits form the
decimal value of a single byte. The non-digit and the three
digits are not considered for mapping. This works the same
way as in Insert mode (see above, i_CTRL-V).
For special keys, the CTRL modifier may be included into the
key to produce a control character. If there is no control
character for the key then its key-notation is inserted.
Note: Under Windows CTRL-V is often mapped to paste text.
Use CTRL-Q instead then.

c_CTRL-Q

CTRL-Q Same as CTRL-V. But with some terminals it is used for
control flow, it doesn't work then.

CTRL-SHIFT-V c_CTRL-SHIFT-V c_CTRL-SHIFT-Q
CTRL-SHIFT-Q Works just like CTRL-V, but do not try to include the CTRL
modifier into the key.
Note: When CTRL-SHIFT-V is intercepted by your system (e.g.,
to paste text) you can often use CTRL-SHIFT-Q instead.
However, in some terminals (e.g. GNOME Terminal), CTRL-SHIFT-Q
quits the terminal without confirmation.

c_<Left> c_Left
<Left> cursor left. See 'wildmenu' for behavior during wildmenu
completion mode.
c_<Right> c_Right
<Right> cursor right. See 'wildmenu' for behavior during wildmenu
completion mode.

c_<S-Left>

<S-Left> or <C-Left> c_<C-Left>

cursor one WORD left

c_<S-Right>

<S-Right> or <C-Right> c_<C-Right>

cursor one WORD right
CTRL-B or <Home> c_CTRL-B c_<Home> c_Home
cursor to beginning of command-line
CTRL-E or <End> c_CTRL-E c_<End> c_End
cursor to end of command-line. See 'wildmenu' for behavior
during wildmenu completion mode.

c_<LeftMouse>

<LeftMouse> Move the cursor to the position of the mouse click.

c_<MiddleMouse>

<MiddleMouse> Paste the contents of the clipboard (for X11 the primary
selection). This is similar to using CTRL-R *, but no CR
characters are inserted between lines.
CTRL-H c_<BS> c_CTRL-H c_BS
<BS> Delete the character in front of the cursor.
c_<Del> c_Del
<Del> Delete the character under the cursor (at end of line:
character before the cursor).

c_CTRL-W

CTRL-W Delete the word before the cursor. This depends on the
'iskeyword' option.

c_CTRL-U

CTRL-U Remove all characters between the cursor position and
the beginning of the line. Previous versions of Vim
deleted all characters on the line. If that is the
preferred behavior, add the following to your vimrc:

```
:cnoremap <C-U> <C-E><C-U>
```

c_<Insert> c_Insert
<Insert> Toggle between insert and overstrike.
{char1} <BS> {char2} or c_digraph

CTRL-K {char1} {char2} c_CTRL-K

enter digraph (see digraphs). When {char1} is a special
key, the code for that key is inserted in <> form.
CTRL-R {register} c_CTRL-R c_<C-R>
Insert the contents of a numbered or named register. Between
typing CTRL-R and the second character '"' will be displayed
to indicate that you are expected to enter the name of a
register.
When used with named or clipboard registers (A-Z,a-z,0-9,+)
text is inserted literally like pasting with "p". For other
registers, the text is inserted as if you typed it, but
mappings and abbreviations are not used. Command-line
completion through 'wildchar' is not triggered though. And
characters that end the command line are inserted literally
(<Esc>, <CR>, <NL>, <C-C>). A <BS> or CTRL-W could still end
the command line though, and remaining characters will then be
interpreted in another mode, which might not be what you
intended.
Special registers:
'"' the unnamed register, containing the text of
the last delete or yank
'%' the current file name
'#' the alternate file name
"*" the clipboard contents (X11: primary
selection)
'+' the clipboard contents
'/' the last search pattern
':' the last command-line

'-'	the last small (less than a line) delete
'.'	the last inserted text

c_CTRL-R_=

'='	the expression register: you are prompted to
enter an expression (see expression)
(doesn't work at the expression prompt; some
things such as changing the buffer or current
window are not allowed to avoid side effects)
When the result is a List the items are used
as lines. They can have line breaks inside
too.
When the result is a Float it's automatically
converted to a String.
Note that when you only want to move the
cursor and not insert anything, you must make
sure the expression evaluates to an empty
string. E.g.:

```
<C-R><C-R>=setcmdpos(2)[-1]<CR>
```

You can use this to insert a register as
typed with CTRL-R =@reg.
See registers about registers.
Implementation detail: When using the expression register
and invoking setcmdpos(), this sets the position before
inserting the resulting string. Use CTRL-R CTRL-R to set the
position afterwards.

CTRL-R CTRL-F	c_CTRL-R_CTRL-F c_<C-R>_<C-F>
CTRL-R CTRL-P	c_CTRL-R_CTRL-P c_<C-R>_<C-P>
CTRL-R CTRL-W	c_CTRL-R_CTRL-W c_<C-R>_<C-W>
CTRL-R CTRL-A	c_CTRL-R_CTRL-A c_<C-R>_<C-A>
CTRL-R CTRL-L	c_CTRL-R_CTRL-L c_<C-R>_<C-L>
Insert the object under the cursor:
CTRL-F the Filename under the cursor
CTRL-P the Filename under the cursor, expanded with
'path' as in gf
CTRL-W the Word under the cursor
CTRL-A the WORD under the cursor; see WORD
CTRL-L the line under the cursor

When 'incsearch' is set the cursor position at the end of the
currently displayed match is used. With CTRL-W the part of
the word that was already typed is not inserted again.

c_CTRL-R_CTRL-R c_<C-R>_<C-R>
c_CTRL-R_CTRL-O c_<C-R>_<C-O>
CTRL-R CTRL-R {register CTRL-F CTRL-P CTRL-W CTRL-A CTRL-L}
CTRL-R CTRL-O {register CTRL-F CTRL-P CTRL-W CTRL-A CTRL-L}
Insert register or object under the cursor. Works like
c_CTRL-R but inserts the text literally. For example, if
register a contains "xy^Hz" (where ^H is a backspace),
"CTRL-R a" will insert "xz" while "CTRL-R CTRL-R a" will
insert "xy^Hz".

CTRL-\ e {expr}	c_CTRL-_e

Evaluate {expr} and replace the whole command line with the
result. You will be prompted for the expression, type <Enter>
to finish it. It's most useful in mappings though. See
expression.
See c_CTRL-R_= for inserting the result of an expression.
Useful functions are getcmdtype(), getcmdline() and
getcmdpos().
The cursor position is unchanged, except when the cursor was
at the end of the line, then it stays at the end.
setcmdpos() can be used to set the cursor position.
The sandbox is used for evaluating the expression to avoid
nasty side effects.
Example:

```
:cmap <F7> <C-\>eAppendSome()<CR>
:func AppendSome()
   :let cmd = getcmdline() .. " Some()"
   :" place the cursor on the )
   :call setcmdpos(strlen(cmd))
   :return cmd
:endfunc
```

This doesn't work recursively, thus not when already editing
an expression. But it is possible to use in a mapping.

c_CTRL-Y

CTRL-Y	When there is a modeless selection, copy the selection into
the clipboard.
If there is no selection CTRL-Y is inserted as a character.
See 'wildmenu' for behavior during wildmenu completion mode.

c_CTRL-Z

CTRL-Z	Trigger 'wildmode'. Same as 'wildcharm', but always available.

CTRL-M or CTRL-J c_CTRL-M c_CTRL-J c_<NL> c_<CR> c_CR
<CR> or <NL> start entered command

CTRL-[c_CTRL-[c_<Esc> c_Esc
<Esc> When typed and 'x' not present in 'cpoptions', quit Command-line mode without executing. In macros or when 'x' present in 'cpoptions', start entered command.
Note: If your <Esc> key is hard to hit on your keyboard, train yourself to use CTRL-[.
c_META c_ALT
ALT (META) may act like <Esc> if the chord is not mapped. For example <A-x> acts like <Esc>x if <A-x> does not have a command-line mode mapping.

c_CTRL-C

CTRL-C quit command-line without executing

c_<Up> c_Up
<Up> recall older command-line from history, whose beginning matches the current command-line (see below). See 'wildmenu' for behavior during wildmenu completion mode.
c_<Down> c_Down
<Down> recall more recent command-line from history, whose beginning matches the current command-line (see below). See 'wildmenu' for behavior during wildmenu completion mode.

c_<S-Up> c_<PageUp>
<S-Up> or <PageUp>
  recall older command-line from history
c_<S-Down> c_<PageDown>
<S-Down> or <PageDown>
  recall more recent command-line from history

CTRL-D command-line completion (see cmdline-completion)
'wildchar' option
  command-line completion (see cmdline-completion)
CTRL-N command-line completion (see cmdline-completion)
CTRL-P command-line completion (see cmdline-completion)
CTRL-A command-line completion (see cmdline-completion)
CTRL-L command-line completion (see cmdline-completion)

c_CTRL-^

CTRL-^ Toggle the use of language :lmap mappings and/or Input Method.
When typing a pattern for a search command and 'imsearch' is not -1, VAL is the value of 'imsearch', otherwise VAL is the value of 'iminsert'.
When language mappings are defined:
- If VAL is 1 (langmap mappings used) it becomes 0 (no langmap mappings used).
- If VAL was not 1 it becomes 1, thus langmap mappings are enabled.

When no language mappings are defined:
- If VAL is 2 (Input Method is used) it becomes 0 (no input method used)
- If VAL has another value it becomes 2, thus the Input Method is enabled.

These language mappings are normally used to type characters that are different from what the keyboard produces. The 'keymap' option can be used to install a whole number of them.
When entering a command line, langmap mappings are switched off, since you are expected to type a command. After switching it on with CTRL-^, the new state is not used again for the next command or Search pattern.

c_CTRL-]

CTRL-] Trigger abbreviation, without inserting a character.

For Emacs-style editing on the command-line see emacs-keys.

The <Up> and <Down> keys take the current command-line as a search string. The beginning of the next/previous command-lines are compared with this string. The first line that matches is the new command-line. When typing these two keys repeatedly, the same string is used again. For example, this can be used to find the previous substitute command: Type ":s" and then <Up>. The same could be done by typing <S-Up> a number of times until the desired command-line is shown. (**Note:** the shifted arrow keys do not work on all terminals)

:his :history
:his[tory] Print the history of last entered commands.

:his[tory] [{name}] [{first}][, [{last}]]
  List the contents of history {name} which can be:
  c[md] or : command-line history

s[earch] or / or ?   search string history
e[xpr]   or =   expression register history
i[nput] or @   input line history
d[ebug] or >   debug command history
a[ll]   all of the above

If the numbers {first} and/or {last} are given, the respective range of entries from a history is listed. These numbers can be specified in the following form:

:history-indexing

A positive number represents the absolute index of an entry as it is given in the first column of a :history listing. This number remains fixed even if other entries are deleted. (see E1510)

A negative number means the relative position of an entry, counted from the newest entry (which has index -1) backwards.

Examples:
List entries 6 to 12 from the search history:

```
:history / 6,12
```

List the penultimate entry from all histories:

```
:history all -2
```

List the most recent two entries from all histories:

```
:history all -2,
```

:keepp[atterns] {command} :keepp :keeppatterns
Execute {command}, without adding anything to the search history and, in case of :s or :&, without modifying the last substitute pattern or substitute string.

2. Command-line completion cmdline-completion

When editing the command-line, a few commands can be used to complete the word before the cursor. This is available for:

- Command names: At the start of the command-line.
- ++opt values.
- Tags: Only after the ":tag" command.
- File names: Only after a command that accepts a file name or a setting for an option that can be set to a file name. This is called file name completion.
- Shell command names: After ":!cmd", ":r !cmd" and ":w !cmd". $PATH is used.
- Options: Only after the ":set" command.
- Mappings: Only after a ":map" or similar command.
- Variable and function names: Only after a ":if", ":call" or similar command.

The number of help item matches is limited (currently to 300) to avoid a long delay when there are very many matches.

For automatic completion as you type (without pressing a key like <Tab>), see cmdline-autocompletion.

These are the commands that can be used:

c_CTRL-D

CTRL-D List names that match the pattern in front of the cursor. When showing file names, directories are highlighted (see highlight-groups). Names where 'suffixes' matches are moved to the end.
The 'wildoptions' option can be set to "tagfile" to list the file of matching tags.

c_CTRL-I c_wildchar c_<Tab> /_<Tab>

'wildchar' option
A match is done on the pattern in front of the cursor. The match (if there are several, the first match) is inserted in place of the pattern. (**Note:** does not work inside a macro, because <Tab> or <Esc> are mostly used as 'wildchar', and these have a special meaning in some macros.) When typed again and there were multiple matches, the next

match is inserted. After the last match, the first is used
again (wrap around).

In search context use <CTRL-V><Tab> or "\t" to search for a
literal <Tab> instead of triggering completion.

The behavior can be changed with the 'wildmode' option.

c_<S-Tab>

<S-Tab> Like 'wildchar' or <Tab>, but begin with the last match and
then go to the previous match.

c_CTRL-N

CTRL-N After using 'wildchar' which got multiple matches, go to next
match. Otherwise recall more recent command-line from
history.

c_CTRL-P

CTRL-P After using 'wildchar' which got multiple matches, go to
previous match. Otherwise recall older command-line from
history.

c_CTRL-A

CTRL-A All names that match the pattern in front of the cursor are
inserted.

c_CTRL-L

CTRL-L A match is done on the pattern in front of the cursor. If
there is one match, it is inserted in place of the pattern.
If there are multiple matches the longest common part is
inserted in place of the pattern. If the result is shorter
than the pattern, no completion is done.

/_CTRL-L

When 'incsearch' is set, entering a search pattern for "/" or
"?" and the current match is displayed then CTRL-L will add
one character from the end of the current match. If
'ignorecase' and 'smartcase' are set and the command line has
no uppercase characters, the added character is converted to
lowercase.

c_CTRL-G /_CTRL-G

CTRL-G While the "/" or "?" 'incsearch' prompt is active, CTRL-G
moves to the next match (without changing the jumplist).
The search-offset is applied when <CR> is pressed, but does
not affect the match highlighting.

Use CTRL-T to move to the previous match.
Mnemonic: on a regular keyboard G is below T.

c_CTRL-T /_CTRL-T

CTRL-T While the "/" or "?" 'incsearch' prompt is active, CTRL-T
moves to the previous match (without changing the jumplist).
The search-offset is applied when <CR> is pressed, but does
not affect the match highlighting.

Use CTRL-G to move to the next match.
Mnemonic: on a regular keyboard T is above G.

The 'wildchar' option defaults to <Tab> (CTRL-E when in Vi compatible mode; in
a previous version <Esc> was used). In the pattern standard wildcards are
accepted when matching file names.

When repeating 'wildchar' or CTRL-N you cycle through the matches, eventually
ending up back to what was typed. If the first match is not what you wanted,
you can use <S-Tab> or CTRL-P to go straight back to what you typed.

The 'wildmenu' option can be set to show the matches just above the command
line.

The 'wildoptions' option provides additional configuration to use a popup menu
for 'wildmenu', and to use fuzzy matching.

The 'wildignorecase' option can be set to ignore case in filenames. For
completing other texts (e.g. command names), the 'ignorecase' option is used
instead (fuzzy matching always ignores case, however).

If you like tcsh's autolist completion, you can use this mapping:

```
:cnoremap X <C-L><C-D>
```

(Where X is the command key to use, <C-L> is CTRL-L and <C-D> is CTRL-D)
This will find the longest match and then list all matching files.

If you like tcsh's autolist completion, you can use the 'wildmode' option to
emulate it. For example, this mimics autolist=ambiguous:

```
:set wildmode=longest,list
```

This will find the longest match with the first 'wildchar', then list all
matching files with the next.

complete-script-local-functions

When completing user function names, prepend "s:" to find script-local
functions.

suffixes

For file name completion you can use the 'suffixes' option to set a priority
between files with almost the same name. If there are multiple matches,
those files with an extension that is in the 'suffixes' option are ignored.
The default is ".bak,~,.o,.h,.info,.swp,.obj", which means that files ending
in ".bak", "~", ".o", ".h", ".info", ".swp" and ".obj" are sometimes ignored.

An empty entry, two consecutive commas, match a file name that does not
contain a ".", thus has no suffix. This is useful to ignore "prog" and prefer
"prog.c".

Examples:

pattern:	files:	match:
test*	test.c test.h test.o	test.c
test*	test.h test.o	test.h and test.o
test*	test.i test.h test.c	test.i and test.c

It is impossible to ignore suffixes with two dots.

If there is more than one matching file (after ignoring the ones matching
the 'suffixes' option) the first file name is inserted. You can see that
there is only one match when you type 'wildchar' twice and the completed
match stays the same. You can get to the other matches by entering
'wildchar', CTRL-N or CTRL-P. All files are included, also the ones with
extensions matching the 'suffixes' option.

To completely ignore files with some extension use 'wildignore'.

To match only files that end at the end of the typed text append a "$". For
example, to match only files that end in ".c":

```
:e *.c$
```

This will not match a file ending in ".cpp". Without the "$" it does match.

If you would like using <S-Tab> for CTRL-P in an xterm, put this command in
your .cshrc:

```
xmodmap -e "keysym Tab = Tab Find"
```

And this in your vimrc:

```
:cmap <Esc>[1~ <C-P>
```

complete-set-option

When setting an option using :set=, the old value of an option can be
obtained by hitting 'wildchar' just after the '='. For example, typing
'wildchar' after ":set dir=" will insert the current value of 'dir'. This
overrules file name completion for the options that take a file name.

When using :set=, :set+=, or :set^=, string options that have
pre-defined names or syntax (e.g. 'diffopt', 'listchars') or are a list of
single-character flags (e.g. 'shortmess') will also present a list of possible
values for completion when using 'wildchar'.

When using :set-=, comma-separated options like 'diffopt' or 'backupdir'
will show each item separately. Flag list options like 'shortmess' will show
both the entire old value and the individual flags. Otherwise completion will
just fill in with the entire old value.

3. Ex command-lines

cmdline-lines

The Ex commands have a few specialties:

:quote :comment

'"' at the start of a line causes the whole line to be ignored. '"'
after a command causes the rest of the line to be ignored. This can be used
to add comments. Example:

```
:set ai         "set 'autoindent' option
```

It is not possible to add a comment to a shell command ":!cmd" or to the
":map" command and a few others (mainly commands that expect expressions)
that see the '"' as part of their argument:

    :argdo
    :autocmd
    :bufdo
    :cexpr (and the like)
    :cdo (and the like)
    :command
    :debug
    :display
    :echo (and the like)
    :elseif
    :execute

    :folddoopen
    :folddoclosed
    :for
    :grep (and the like)
    :help (and the like)
    :if
    :let
    :make
    :map (and the like including :abbrev commands)
    :menu (and the like)
    :mkspell
    :normal
    :popup
    :registers
    :return
    :sort
    :syntax
    :tabdo
    :tearoff
    :vimgrep (and the like)
    :while
    :windo

:bar :\bar

'|' can be used to separate commands, so you can give multiple commands in one line. If you want to use '|' in an argument, precede it with '\'.

These commands see the '|' as their argument, and can therefore not be followed by another Vim command:

    :append
    :argdo
    :autocmd
    :bufdo
    :cdo
    :cfdo
    :change
    :command
    :debug
    :eval
    :folddoopen
    :folddoclosed
    :function
    :global
    :help
    :helpgrep
    :insert
    :ldo
    :lfdo
    :lhelpgrep
    :lsp
    :make
    :normal
    :perlfile
    :pyfile
    :python
    :registers
    :read !
    :sign
    :tabdo
    :terminal
    :vglobal
    :windo
    :write !
    :[range]!
    a user defined command without the "-bar" argument :command

Note that this is confusing (inherited from Vi): With ":g" the '|' is included in the command, with ":s" it is not.

To be able to use another command anyway, use the ":execute" command.
Example (append the output of "ls" and jump to the first line):

```
:execute 'r !ls' | '[
```

There is one exception: When the 'b' flag is present in 'cpoptions', with the ":map" and ":abbr" commands and friends CTRL-V needs to be used instead of '\'. You can also use "<Bar>" instead. See also map_bar.

Examples:

```
:!ls | wc               view the output of two commands
:r !ls | wc             insert the same output in the text
:%g/foo/p|>             moves all matching lines one shiftwidth
:%s/foo/bar/|>          moves one line one shiftwidth
:map q 10^V|            map "q" to "10|"
:map q 10\| map \ l     map "q" to "10\" and map "\" to "l"
                                (when 'b' is present in 'cpoptions')
```

You can also use <NL> to separate commands in the same way as with '|'. To insert a <NL> use CTRL-V CTRL-J. "^@" will be shown. Using '|' is the

preferred method. But for external commands a <NL> must be used, because a
'|' is included in the external command. To avoid the special meaning of <NL>
it must be preceded with a backslash. Example:

```
:r !date<NL>-join
```

This reads the current date into the file and joins it with the previous line.

Note that when the command before the '|' generates an error, the following
commands will not be executed.

Because of Vi compatibility the following strange commands are supported:

```
:|                      print current line (like ":p")
:3|                     print line 3 (like ":3p")
:3                      goto line 3
```

A colon is allowed between the range and the command name. It is ignored
(this is Vi compatible). For example:

```
:1,$:s/pat/string
```

When the character '%' or '#' is used where a file name is expected, they are
expanded to the current and alternate file name (see the chapter "editing
files" :_% :_#).

Trailing spaces in filenames will be ignored, unless escaped with a backslash
or CTRL-V. Note that the ":next" command uses spaces to separate file names.
Escape the spaces to include them in a file name. Example:

```
:next foo\ bar goes\ to school\
```

starts editing the three files "foo bar", "goes to" and "school ".

When you want to use the special characters '"' or '|' in a command, or want
to use '%' or '#' in a file name, precede them with a backslash. The
backslash is not required in a range and in the ":substitute" command.
See also `:=.

:_!

The '!' (bang) character after an Ex command makes the command behave in a
different way. The '!' should be placed immediately after the command,
without any blanks in between. If you insert blanks the '!' will be seen as
an argument for the command, which has a different meaning. For example:

```
        :w! name        write the current buffer to file "name", overwriting
                        any existing file
        :w !name        send the current buffer as standard input to command
                        "name"
```

4. Ex command-line ranges

cmdline-ranges [range] E16

Some Ex commands accept a line range in front of them. This is noted as
[range]. It consists of one or more line specifiers, separated with ',' or
';'.

The basics are explained in section 10.3 of the user manual.

:, :;

When separated with ';' the cursor position will be set to that line
before interpreting the next line specifier. This doesn't happen for ','.
Examples:

```
4,/this line/
```

        from line 4 till match with "this line" after the cursor line.

```
5;/that line/
```

        from line 5 till match with "that line" after line 5.

The default line specifier for most commands is the cursor position, but the
commands ":write" and ":global" have the whole buffer (1,$) as default.

If more line specifiers are given than required for the command, when comma
separated, the leftmost one(s) will be ignored, e.g., the -2,+ in this:

```
:-2,+,-2,. print
```

<When semicolon separated, the leftmost specifier to the penultimate one are
summed, e.g., -4 + 3 - 1 = -2, in this:

```
:-4;+3;-1;+2 print
```

Line numbers may be specified with: :range {address}

```
        {number}        an absolute line number  E1247
        .               the current line                                    :.

        $               the last line of the buffer                         :$
```

```
%               equal to 1,$ (the entire buffer)                                    :%

*               equal to '<,'> (the lines of the last
                selected Visual area; see :star below)
'x              the line of the position of mark x                                  :'x

                (where x is any {a-z} mark)
'X              the line of the position of mark X                                  :'X

                (where X is any {A-Z0-9} mark, though
                when X is in another buffer it cannot
                be used in a range)
'[              the first line of the most recent                                   :'[

                change or yank
']              the last line of the most recent                                    :']

                change or yank
'<              the first line of the most recently                                 :'<

                selected Visual area
'>              the last line of the most recently                                  :'>

                selected Visual area
''              the line of the position before the                                 :''

                latest jump, or where the last "m'"/"m`"
                command was given (though '' is 1 if it
                isn't in the current buffer)
'"              the line of the cursor position when                                :'quote

                last exiting the buffer
'^              the line of the cursor position the                                 :'^

                last time Insert mode was stopped
'.              the line of the cursor position when the   :'.
                buffer was last changed
'(              the line of the first character of the                              :'(

                current sentence
')              the line of the first character after                               :')

                the end of the current sentence
'{              the first empty line before the                                     :'{

                paragraph containing the cursor
'}              the first empty line after the                                      :'}

                paragraph containing the cursor
/{pattern}[/]   the next line where {pattern} matches                               :/

                        also see :range-pattern below
?{pattern}[?]   the previous line where {pattern} matches  :?
                        also see :range-pattern below
\/              the next line where the most recent
                search pattern matches
\?              the previous line where the most recent
                search pattern matches
\&              the next line where the most recent
                substitute pattern matches
```

Note: "next line" and "previous line" do not include matches appearing in the current line.

:range-offset

Each line specifier may be followed by one or more '+' or '-' and an optional number. That value is added or subtracted from the preceding line number. So, for example, 'x+2 is two lines after the line containing mark x. If the number is omitted, +1 is used for each '+' and -1 for each '-' so, e.g., 'x++ and 'x+2 are synonymous. If there is nothing before the '+' or '-', for the first line number in [range] the current line is used as the relative starting point. So, -,. means, "the line before the current line to the current line". The value of the second line number in [range] depends on whether a comma or semicolon separates the line numbers (see :, and :;). Examples: If the cursor is within the line below this one, any of these commands will print the tag line ":range-offset" and the line, "Each...":

```
:-11;+1 print
:-----------,-10 print
:?Each line?-;+ print
:'{+,'{+2 print
:'{+1;')-1 print
```

:range-closed-fold

When a line number after the comma is in a closed fold it is adjusted to the last line of the fold, thus the whole fold is included.

When a number is added this is done after the adjustment to the last line of the fold. This means these lines are additionally included in the range. For

example:

```
:3,4+2print
```

On this text:
1 one
2 two
3 three
4 four FOLDED
5 five FOLDED
6 six
7 seven
8 eight
Where lines four and five are a closed fold, ends up printing lines 3 to 7.
The 7 comes from the "4" in the range, which is adjusted to the end of the
closed fold, which is 5, and then the offset 2 is added.

An example for subtracting (which isn't very useful):

```
:2,4-1print
```

On this text:
1 one
2 two
3 three FOLDED
4 four FOLDED
5 five FOLDED
6 six FOLDED
7 seven
8 eight
Where lines three to six are a closed fold, ends up printing lines 2 to 6.
The 6 comes from the "4" in the range, which is adjusted to the end of the
closed fold, which is 6, and then 1 is subtracted, then this is still in the
closed fold and the last line of that fold is used, which is 6.

:range-pattern

The "/" and "?" after {pattern} are required to separate the pattern from
anything that follows.

The "/" and "?" may be preceded with another address. The search starts from
there. The difference from using ';' is that the cursor isn't moved.
Examples:

```
/pat1//pat2/    Find line containing "pat2" after line containing
                "pat1", without moving the cursor.
7;/pat2/        Find line containing "pat2", after line 7, leaving
                the cursor in line 7.
```

The {number} must be between 0 and the number of lines in the file. When
using a 0 (zero) this is interpreted as a 1 by most commands. Commands that
use it as a count do use it as a zero (:tag, :pop, etc). Some commands
interpret the zero as "before the first line" (:read, search pattern, etc).

Examples:

```
.+3             three lines below the cursor
/that/+1        the line below the next line containing "that"
.,$             from current line until end of file
0;/that         the first line containing "that", also matches in the
                first line.
1;/that         the first line after line 1 containing "that"
```

Some commands allow for a count after the command. This count is used as the
number of lines to be used, starting with the line given in the last line
specifier (the default is the cursor line). The commands that accept a count
are the ones that use a range but do not have a file name argument (because
a file name can also be a number). The count cannot be negative.

Examples:

```
:s/x/X/g 5      substitute 'x' by 'X' in the current line and four
                following lines
:23d 4          delete lines 23, 24, 25 and 26
```

Folds and Range

When folds are active the line numbers are rounded off to include the whole
closed fold. See fold-behavior.

Reverse Range E493

A range should have the lower line number first. If this is not the case, Vim
will ask you if it should swap the line numbers.
Backwards range given, OK to swap
This is not done within the global command ":g".

You can use ":silent" before a command to avoid the question, the range will
always be swapped then.

Count and Range N:

When giving a count before entering ":", this is translated into:

```
:.,.+(count - 1)
```

In words: The "count" lines at and after the cursor. Example: To delete three lines:

```
3:d<CR>         is translated into: .,.+2d<CR>
```

Visual Mode and Range

v_:

{Visual}: Starts a command-line with the Visual selected lines as a range. The code :'<,'> is used for this range, which makes it possible to select a similar line from the command-line history for repeating a command on different Visually selected lines.

:* :star :star-visual-range
When Visual mode was already ended, a short way to use the Visual area for a range is :*.

5. Ex command-line flags ex-flags

These flags are supported by a selection of Ex commands. They print the line that the cursor ends up after executing the command:

l	output like for :list
#	add line number
p	output like for :print

The flags can be combined, thus "l#" uses both a line number and :list style output.

6. Ex special characters cmdline-special

Note: These are special characters in the executed command line. If you want to insert special things while typing you can use the CTRL-R command. For example, "%" stands for the current file name, while CTRL-R % inserts the current file name right away. See c_CTRL-R.

Note: If you want to avoid the effects of special characters in a Vim script you may want to use fnameescape(). Also see `=.

In Ex commands, at places where a file name can be used, the following characters have a special meaning. These can also be used in the expression function expand().

%	Is replaced with the current file name.	:_% c_%
#	Is replaced with the alternate file name. This is remembered for every window.	:_# c_#
#n	(where n is a number) is replaced with the file name of buffer n. "#0" is the same as "#".	:_#0 :_#n c_#n
##	Is replaced with all names in the argument list concatenated, separated by spaces. Each space in a name is preceded with a backslash.	:_## c_##
#<n	(where n is a number > 0) is replaced with old file name n. See :oldfiles or v:oldfiles to get the number.	:_#< c_#< E809

Note that these, except "#<n", give the file name as it was typed. If an absolute path is needed (when using the file name from a different directory), you need to add ":p". See filename-modifiers.

The "#<n" item returns an absolute path, but it will start with "~/" for files below your home directory.

Note that backslashes are inserted before spaces, so that the command will correctly interpret the file name. But this doesn't happen for shell commands. For those you probably have to use quotes (this fails for files that contain a quote and wildcards):

```
:!ls "%"
:r !spell "%"
```

To avoid the special meaning of '%' and '#' insert a backslash before it.
Detail: The special meaning is always escaped when there is a backslash before it, no matter how many backslashes.

you type:	result
#	alternate.file
\#	#
\\#	\#

Also see `=.

E499 E500

Note: these are typed literally, they are not special keys!

:<cword> <cword>
<cword> is replaced with the word under the cursor (like star)

:<cWORD> <cWORD>
<cWORD> is replaced with the WORD under the cursor (see WORD)

:<cexpr> <cexpr>
<cexpr> is replaced with the word under the cursor, including more

to form a C expression. E.g., when the cursor is on "arg" of "ptr->arg" then the result is "ptr->arg"; when the cursor is on "]" of "list[idx]" then the result is "list[idx]".

:<cfile> <cfile>

<cfile> is replaced with the path name under the cursor (like what gf uses)

:<afile> <afile>

<afile> When executing autocommands, is replaced with the file name of the buffer being manipulated, or the file for a read or write. E495

:<abuf> <abuf>

<abuf> When executing autocommands, is replaced with the currently effective buffer number. It is not set for all events, also see bufnr(). For ":r file" and ":so file" it is the current buffer, the file being read/sourced is not in a buffer. E496

:<amatch> <amatch>

<amatch> When executing autocommands, is replaced with the match for which this autocommand was executed. E497
It differs from <afile> when the file name isn't used to match with (for FileType, Syntax and SpellFileMissing events).
When the match is with a file name, it is expanded to the full path.

:<stack> <stack>

<stack> is replaced with the call stack, using "function {function-name}[{lnum}]" for a function line and "script {file-name}[{lnum}]" for a script line, and ".." in between items. E.g.:
"function {function-name1}[{lnum}]..{function-name2}[{lnum}]"
If there is no call stack you get error E489 .

:<script> <script> E498

<script> When executing a :source command, is replaced with the file name of the sourced file. When executing a function, is replaced with the file name of the script where it is defined.
If the file name cannot be determined you get error E1274 .

:<slnum> <slnum>

<slnum> When executing a :source command, is replaced with the line number. E842
When executing a function it's the line number relative to the start of the function.

:<sflnum> <sflnum>

<sflnum> When executing a script, is replaced with the line number. It differs from <slnum> in that <sflnum> is replaced with the script line number in any situation. E961

filename-modifiers

:_%: ::8 ::p ::. ::~ ::h ::t ::r ::e ::s ::gs ::S
%:8 %:p %:. %:~ %:h %:t %:r %:e %:s %:gs %:S

The file name modifiers can be used after "%", "#", "#n", "<cfile>", "<script>", "<afile>" or "<abuf>". They are also used with the fnamemodify() function.

These modifiers can be given, in this order:

:p Make file name a full path. Must be the first modifier. Also changes "~/" (and "~user/" for Unix) to the path for the home directory. If the name is a directory a path separator is added at the end. For a file name that does not exist and does not have an absolute path the result is unpredictable. On MS-Windows an 8.3 filename is expanded to the long name.

:8 Converts the path to 8.3 short format (currently only on MS-Windows). Will act on as much of a path that is an existing path.

:~ Reduce file name to be relative to the home directory, if possible. File name is unmodified if it is not below the home directory.

:. Reduce file name to be relative to current directory, if possible. File name is unmodified if it is not below the current directory.
For maximum shortness, use ":~:.".

:h Head of the file name (the last component and any separators removed). Cannot be used with :e, :r or :t.
Can be repeated to remove several components at the end.
When the file name ends in a path separator, only the path separator is removed. Thus ":p:h" on a directory name results on the directory name itself (without trailing slash).
When the file name is an absolute path (starts with "/" for Unix; "x:\" for Win32), that part is not removed.
When there is no head (path is relative to current directory) the result is empty.

:t Tail of the file name (last component of the name). Must precede any :r or :e.

:r Root of the file name (the last extension removed). When there is only an extension (file name that starts with '.', e.g., ".nvimrc"), it is not removed. Can be repeated to remove several extensions (last one first).

:e Extension of the file name. Only makes sense when used alone. When there is no extension the result is empty.

When there is only an extension (file name that starts with '.'), the result is empty. Can be repeated to include more extensions. If there are not enough extensions (but at least one) as much as possible are included.

:s?pat?sub?
Substitute the first occurrence of "pat" with "sub". This works like the :s command. "pat" is a regular expression. Any character can be used for '?', but it must not occur in "pat" or "sub".
After this, the previous modifiers can be used again. For example ":p", to make a full path after the substitution.

:gs?pat?sub?
Substitute all occurrences of "pat" with "sub". Otherwise this works like ":s".

:S Escape special characters for use with a shell command (see shellescape()). Must be the last one. Examples:

```
:!dir <cfile>:S
:call system('chmod +w -- ' .. expand('%:S'))
```

Examples, when the file name is "src/version.c", current dir "/home/mool/vim":

```
:p                      /home/mool/vim/src/version.c
:p:.                               src/version.c
:p:~                         ~/vim/src/version.c
:h                                 src
:p:h                    /home/mool/vim/src
:p:h:h          /home/mool/vim
:t                                      version.c
:p:t                                    version.c
:r                                 src/version
:p:r                    /home/mool/vim/src/version
:t:r                                    version
:e                                             c
:s?version?main?                   src/main.c
:s?version?main?:p      /home/mool/vim/src/main.c
:p:gs?/?\\?             \home\mool\vim\src\version.c
```

Examples, when the file name is "src/version.c.gz":

```
:p                      /home/mool/vim/src/version.c.gz
:e                                               gz
:e:e                                            c.gz
:e:e:e                                  c.gz
:e:e:r                                  c
:r                                 src/version.c
:r:e                                            c
:r:r                               src/version
:r:r:r                        src/version
```

extension-removal :_%<

If a "<" is appended to "%", "#", "#n" or "CTRL-V p" the extension of the file name is removed (everything after and including the last '.' in the file name). This is included for backwards compatibility with version 3.0, the ":r" form is preferred. Examples:

```
%               current file name
%<              current file name without extension
#               alternate file name for current window
#<              idem, without extension
#31             alternate file number 31
#31<            idem, without extension
<cword>         word under the cursor
<cWORD>         WORD under the cursor (see |WORD|)
<cfile>         path name under the cursor
<cfile><        idem, without extension
```

Note: Where a file name is expected wildcards expansion is done. On Unix the shell is used for this, unless it can be done internally (for speed). Backticks work also, like in

```
:n `echo *.c`
```

But expansion is only done if there are any wildcards before expanding the '%', '#', etc.. This avoids expanding wildcards inside a file name. If you want to expand the result of <cfile>, add a wildcard character to it. Examples: (alternate file name is "?readme?")

command expands to

```
:e #            :e ?readme?
:e `ls #`       :e {files matching "?readme?"}
:e #.*          :e {files matching "?readme?.*"}
:cd <cfile>     :cd {file name under cursor}
:cd <cfile>*    :cd {file name under cursor plus "*" and then expanded}
```

Also see `=.

When the expanded argument contains a "!" and it is used for a shell command (":!cmd", ":r !cmd" or ":w !cmd"), the "!" is escaped with a backslash to avoid it being expanded into a previously used command. When the 'shell' option contains "sh", this is done twice, to avoid the shell trying to expand the "!".

filename-backslash

For filesystems that use a backslash as directory separator (Windows filesystems), it's a bit difficult to recognize a backslash that is used to escape the special meaning of the next character. The general rule is: If the backslash is followed by a normal file name character, it does not have a special meaning. Therefore "\file\foo" is a valid file name, you don't have to type the backslash twice.

An exception is the '$' sign. It is a valid character in a file name. But to avoid a file name like "$home" to be interpreted as an environment variable, it needs to be preceded by a backslash. Therefore you need to use "/\$home" for the file "$home" in the root directory. A few examples:

FILE NAME	INTERPRETED AS
$home	expanded to value of environment var $home
\$home	file "$home" in current directory
/\$home	file "$home" in root directory
\\$home	file "\\", followed by expanded $home

Also see `=.

7. Command-line window

cmdline-window cmdwin

command-line-window

The command-line window ("cmdwin") is a buffer+window in which the command line can be edited. See also :exmode, a REPL-like mode based on the cmdwin.

OPEN

c_CTRL-F q: q/ q?

There are two ways to open the command-line window:

1. From Command-line mode, use <C-F> (or the 'cedit' key).

   - Not supported from the expr-register cmdline (c_CTRL-R_= i_CTRL-R_=).

2. From Normal mode, use the "q:", "q/" or "q?" command.

   This starts editing an Ex command-line ("q:") or search string ("q/" or "q?"). Note that this is not possible while recording is in progress (the "q" stops recording then).

When the window opens it is filled with the command-line history. The editor opens in Normal mode. The last line contains the command as typed so far. The cmdwin-char is shown in 'statuscolumn', indicating the type of command-line. The window is full width, positioned just above the command-line; its height is decided by 'cmdwinheight' (if there is room).

Window and tabpage commands work as usual while cmdwin is open: you can navigate, resize, close and open windows/tabpages. The mouse works as in any window: clicks focus other windows, and statuslines and vertical separators can be dragged to resize.

E1292

Only one command-line window can be open at a time.

EDIT

You can use any commands to move around and edit the text in the window, both in Normal and Insert mode.

It is possible to use ":", "/" and other commands that use the command-line, but it's not possible to open another cmdwin from there. There is no nesting.

CLOSE

There are several ways to leave the command-line window:

<Enter>	Execute the command-line under the cursor, then close the window(s) where cmdwin is visible. Works both in Insert and in Normal mode.
CTRL-C	Continue in Command-line mode. The command-line under the cursor is pre-filled in the command-line. Works both in Insert and in Normal mode.
:quit	Discard the command line and go back to Normal mode. ":close", CTRL-W c, ":exit", ":xit" and CTRL-\ CTRL-N also work.
:qall	Quit Nvim, unless there are changes in some buffer.
:qall!	Quit Nvim, discarding changes to any buffer.

When the cmdwin closes, focus returns to the window where the command-line
was started from (if possible). The cmdwin scratch buffer is destroyed
(bufhidden=wipe). Any text edits other than the line executed by <Enter>
are lost.

If you would like to execute the command under the cursor and then have the
command-line window open again, you may find this mapping useful:

```
:autocmd CmdwinEnter * map <buffer> <F5> <CR>q:
```

VARIOUS

Buffer and window options set on cmdwin open:

'bufhidden'	"wipe"
'buflisted'	off
'buftype'	"nofile"
'filetype'	"vim" (only for ":" type)
'foldenable'	off
'statuscolumn'	shows the cmdwin-char
'swapfile'	off

It is allowed to write the buffer contents to a file. This is an easy way to
save the command-line history and read it back later.

If the 'wildchar' option is set to <Tab>, and the command-line window is used
for an Ex command, then two mappings will be added to use <Tab> for completion
in the command-line window, like this:

```
:inoremap <buffer> <Tab> <C-X><C-V>
:nnoremap <buffer> <Tab> a<C-X><C-V>
```

Note that hitting <Tab> in Normal mode will do completion on the next
character. That way it works at the end of the line.
If you don't want these mappings, disable them with:

```
au CmdwinEnter [:>] iunmap <buffer> <Tab>
au CmdwinEnter [:>] nunmap <buffer> <Tab>
```

You could put these lines in your vimrc file.

The getcmdwintype() function returns the type of the command-line being
edited as described in cmdwin-char.

Nvim defines this default CmdwinEnter autocmd in the "nvim.cmdwin" group:

```
autocmd CmdwinEnter [:>] syntax sync minlines=1 maxlines=1
```

You can disable this in your config with "autocmd! nvim.cmdwin". default-autocmds

AUTOCOMMANDS

Two autocommand events are used: CmdwinEnter and CmdwinLeave. You can use
the Cmdwin events to do settings specifically for the command-line window.
Be careful not to cause side effects!
Example:

```
:au CmdwinEnter :  let b:cpt_save = &cpt | set cpt=.
:au CmdwinLeave :  let &cpt = b:cpt_save
```

This sets 'complete' to use completion in the current window for i_CTRL-N.
Another example:

```
:au CmdwinEnter [\/\?]  startinsert
```

This will make Vim start in Insert mode in the command-line window.
Note: The "/" and "?" needs to be escaped, as this is a file-pattern.
See also cmdline-autocompletion.

cmdline-char cmdwin-char

The character used for the pattern indicates the type of command-line:

:	normal Ex command
>	debug mode command debug-mode
/	forward search string
?	backward search string
=	expression for "= expr-register
@	string for input()
-	text for :insert or :append

8. Command-line autocompletion cmdline-autocompletion

Autocompletion makes the command-line more efficient and easier to navigate by
automatically showing a popup menu of suggestions as you type, whether
searching (/ or ?) or entering commands (:).

A basic setup is:

```
autocmd CmdlineChanged [:\/\?] call wildtrigger()
set wildmode=noselect:lastused,full
set wildoptions=pum
```

With this configuration, suggestions appear immediately, and you can
move through them with <Tab> or the arrow keys.

To retain normal command-line history navigation with <Up>/<Down>:

```
cnoremap <expr> <Up>   wildmenumode() ? "\<C-E>\<Up>"   : "\<Up>"
cnoremap <expr> <Down> wildmenumode() ? "\<C-E>\<Down>" : "\<Down>"
```

Options can also be applied only for specific command-lines. For
example, to use a shorter popup menu height only during search:

```
autocmd CmdlineEnter [\/\?] set pumheight=8
autocmd CmdlineLeave [\/\?] set pumheight&
```

EXTRAS **fuzzy-file-picker** **live-grep**

Command-line autocompletion can also be extended for advanced uses.
For example, you can turn the native :find command into a fuzzy, interactive
file picker:

```
set findfunc=Find
func Find(arg, _)
  if empty(s:filescache)
    let s:filescache = globpath('.', '**', 1, 1)
    call filter(s:filescache, '!isdirectory(v:val)')
    call map(s:filescache, "fnamemodify(v:val, ':.')")
  endif
  return a:arg == '' ? s:filescache : matchfuzzy(s:filescache, a:arg)
endfunc
let s:filescache = []
autocmd CmdlineEnter : let s:filescache = []
```

The :Grep command searches for lines matching a pattern and updates the
results dynamically as you type (triggered after two characters; note: needs
the CmdlineLeavePre autocmd from the next section):

```
command! -nargs=+ -complete=customlist,<SID>Grep
        \ Grep call <SID>VisitFile()

func s:Grep(arglead, cmdline, cursorpos)
  if match(&grepprg, '\$\*') == -1 | let &grepprg ..= ' $*' | endif
  let cmd = substitute(&grepprg, '\$\*', shellescape(escape(a:arglead, '\')), '')
  return len(a:arglead) > 1 ? systemlist(cmd) : []
endfunc

func s:VisitFile()
  let item = getqflist(#{lines: [s:selected]}).items[0]
  let pos  = '[0,\ item.lnum,\ item.col,\ 0]'
  exe $':b +call\ setpos(".",\ {pos}) {item.bufnr}'
  call setbufvar(item.bufnr, '&buflisted', 1)
endfunc
```

Automatically select the first item in the completion list when leaving the
command-line, and for :Grep, add the typed pattern to the command-line
history:

```
autocmd CmdlineLeavePre :
      \ if get(cmdcomplete_info(), 'matches', []) != [] |
      \   let s:info = cmdcomplete_info() |
      \   if getcmdline() =~ '^\s*fin\%[d]\s' && s:info.selected == -1 |
      \     call setcmdline($'find {s:info.matches[0]}') |
      \   endif |
      \   if getcmdline() =~ '^\s*Grep\s' |
      \     let s:selected = s:info.selected != -1
      \         ? s:info.matches[s:info.selected] : s:info.matches[0] |
      \     call setcmdline(s:info.cmdline_orig) |
      \   endif |
      \ endif
```

For autocompletion in insert mode, see ins-autocompletion.

Credits

Most of Vim was written by Bram Moolenaar <Bram@vim.org> Bram-Moolenaar.

Parts of the documentation come from several Vi manuals, written by:
	W.N. Joy
	Alan P.W. Hewett
	Mark Horton

The Vim editor is based on Stevie and includes (ideas from) other software,
worked on by the people mentioned here. Other people helped by sending me
patches, suggestions and giving feedback about what is good and bad in Vim.

Vim would never have become what it is now, without the help of these people!

Bram Moolenaar	Creator and benevolent dictator until his passing (3 August 2023)
Ron Aaron	Win32 GUI changes
Mohsin Ahmed	encryption
Zoltan Arpadffy	work on VMS port
Tony Andrews	Stevie
Gert van Antwerpen	changes for DJGPP on MS-DOS
Berkeley DB(3)	ideas for swap file implementation
Keith Bostic	Nvi
Walter Briscoe	Makefile updates, various patches
Ralf Brown	SPAWNO library for MS-DOS
Robert Colon	many useful remarks
Marcin Dalecki	GTK+ GUI port, toolbar icons, gettext()
Kayhan Demirel	sent me news in Uganda
Chris & John Downey	xvi (ideas for multi-windows version)
Henk Elbers	first VMS port
Daniel Elstner	GTK+ 2 port
Eric Fischer	Mac port, 'cindent', and other improvements
Benji Fisher	Answering lots of user questions
Bill Foster	Athena GUI port (later removed)
Google	Let Bram work on Vim one day a week
Loic Grenie	xvim (ideas for multi windows version)
Sven Guckes	Vim promoter and previous WWW page maintainer
Darren Hiebert	Exuberant ctags
Jason Hildebrand	GTK+ 2 port
Bruce Hunsaker	improvements for VMS port
Andy Kahn	Cscope support, GTK+ GUI port
Oezguer Kesim	Maintainer of Vim Mailing Lists
Axel Kielhorn	work on the Macintosh port
Steve Kirkendall	Elvis
Roger Knobbe	original port to Windows NT
Sergey Laskavy	Vim's help from Moscow
Felix von Leitner	Previous maintainer of Vim Mailing Lists
David Leonard	Port of Python extensions to Unix
Avner Lottem	Edit in right-to-left windows
Flemming Madsen	X11 client-server, various features and patches
Tony Mechelynck	answers many user questions
Paul Moore	Python interface extensions, many patches
Katsuhito Nagano	Work on multibyte versions
Sung-Hyun Nam	Work on multibyte versions
Vince Negri	Win32 GUI and generic console enhancements
Steve Oualline	Author of the first Vim book frombook
Dominique Pelle	Valgrind reports and many fixes
A.Politz	Many bug reports and some fixes
George V. Reilly	Win32 port, Win32 GUI start-off
Stephen Riehm	bug collector
Stefan Roemer	various patches and help to users
Ralf Schandl	IBM OS/390 port
Olaf Seibert	DICE and BeBox version, regexp improvements
Mortaza Shiran	Farsi patches
Peter da Silva	termlib
Paul Slootman	OS/2 port
Henry Spencer	regular expressions
Dany St-Amant	Macintosh port
Tim Thompson	Stevie
G. R. (Fred) Walter	Stevie
Sven Verdoolaege	Perl interface
Robert Webb	Command-line completion, GUI versions, and lots of patches
Ingo Wilken	Tcl interface
Mike Williams	PostScript printing
Juergen Weigert	Lattice version, AUX improvements, Unix and MS-DOS ports, autoconf
Stefan 'Sec' Zehl	Maintainer of vim.org
Yasuhiro Matsumoto	many MS-Windows improvements
Ken Takata	fixes and features
Kazunobu Kuriyama	GTK 3
Christian Brabandt	many fixes, features, user support, etc.
Yegappan Lakshmanan	many quickfix features, Vim9 script improvements
Doug Kearns	Runtime file maintainer
Foxe Chen	Wayland support, new features
glepnir	work on improving completion feature, fixes
Girish Palya	autocompletion (ins/cmdline), omnifunc composing, search/subst completion, and more.

Hirohito Higashi lots of patches and fixes
Yee Cheng Chin MacVim maintainer and diff improvements
zeertzjq many fixes and improvements

I wish to thank all the people that sent me bug reports and suggestions. The list is too long to mention them all here. Vim would not be the same without the ideas from all these people: They keep Vim alive!
love peace friendship gross-national-happiness

Documentation may refer to other versions of Vi:

Vi vi

Vi "the original". Without further remarks this is the version of Vi that appeared in Sun OS 4.x. ":version" returns "Version 3.7, 6/7/85". Source code only available with a license.

Nvi

Nvi The "New" Vi. The version of Vi that comes with BSD 4.4 and FreeBSD. Very good compatibility with the original Vi, with a few extensions. The version used is 1.79. ":version" returns "Version 1.79 (10/23/96)". Source code is freely available.

Elvis

Elvis Another Vi clone, made by Steve Kirkendall. Very compact but isn't as flexible as Vim. Source code is freely available.

Vim Nvim is based on Vim. https://www.vim.org/

Neovim fundraiser backers backers.txt

Thank you to everyone who backed the original Neovim Fundraiser.

LIST OF BACKERS

- [Bob Breznak](http://brez.io)
- [Tim Uruski](http://timuruski.net)
- @mikker (http://brnbw.com)
- Aaron J. [@metaxis](http://twitter.com/metaxis) Peterson
- Adam George http://www.adamgeorge.com
- Adam Lindberg (http://alind.io)
- Adam McLain http://twitter.com/adammclain
- Adam Piper http://ahri.net
- Adrian Philipp http://adrian-philipp.com
- Alexander Quine
- Alexandru Dabija
- Alexis Hildebrandt http://surryhill.net
- Andrew M. Farrell http://amfarrell.com
- Anton Beloglazov http://beloglazov.info/
- Arthur Axel fREW Schmidt - https://blog.afoolishmanifesto.com
- Ben Thouret http://benjamin.thouret.com
- Benedikt Böhm http://bb.xnull.de
- bfredl, http://bfredl.github.io
- Brandon Smith, http://reardencode.com

- Brandon Wang, http://brandonwang.me
- Brennen Bearnes http://p1k3.com
- Brock Wilcox (awwaiid) http://thelackthereof.org/
- Cameron Eagans http://cweagans.net
- Carl Myers http://cmyers.org/
- Carlos Alves http://www.carlosalves.info
- Chris Chernesky, http://www.tinderglow.com
- Chris Hartjes
- Christoffer Holmstedt (http://www.christofferholmstedt.se)
- Ciaran Downey <http://ciarand.me>
- Craig Bane http://standardizedways.co.uk
- Dane Summers http://pinedesk.biz
- Daniel Collin https://twitter.com/daniel_collin
- Daniel Rogers http://remotecodelist.com
- Darshan Sawardekar http://pressing-matters.io
- David Gabriel http://wiki.dest-unreachable.net
- Deepak Kumar www.kreatio.com
- Derek Rodger http://devslant.com
- ebene fünf GmbH www.ebenefuenf.de
- Ehtesh Choudhury
- Emil Hernvall http://emilhernvall.github.io/
- Eric Siegel https://github.com/esiegel
- Ethan Schoonover http://ethanschoonover.com
- FFunction http://ffctn.com
- Frank http://duckpun.ch
- Gabriele Lana http://github.com/gabrielelana
- Gavin Rogers http://praxeology.co.uk
- Geoff Greer https://floobits.com/
- Glenn Strauss https://github.com/gstrauss/
- Gustav Buchholtz http://grod.se

- Henri Bourcereau http://bourcereau.fr
- http://jaredforsyth.com
- http://www.1bsyl.com
- Johann Dahm (http://johanndahm.com)
- Ian Farmer http://farmernet.net
- iauns - http://www.noobtube.com
- Jacob Jewell www.immersiveapplications.com
- James A. Overton http://james.overton.ca
- James Tomasino - http://jamestomasino.com
- Jamiel Almeida http://slashfoo.com
- Jan Christoph Ebersbach http://www.e-jc.de/
- Jason Peacock http://jasonpeacock.com
- Jason Weathered http://jasoncodes.com/
- Javier Maestro
- Jeff Craig http://blog.foxxtrot.net/
- Jeremy Huffman http://jeremyhuffman.com
- Jeremy Morrell http://rathercurio.us
- Jesper Kjeldgaard (thejspr) - http://thejspr.com
- Jim Hester (http://jimhester.com)
- Joe Moon http://joe.xoxomoon.com
- Johan Simonsson, http://fun-tech.se
- John P. Kiffmeyer (http://jpk.is)
- John Szakmeister http://www.szakmeister.net/
- John W. Long http://wiseheartdesign.com Twitter: @johnwlong GitHub: @jlong
- Jon Yurek http://thoughtbot.com
- jonaias http://jonaias.info
- Josh Davis, http://joshldavis.com
- Joshua Levesque (www.jandyco.com)
- Justin M. Keyes https://github.com/justinmk
- Justin Russell http://jrussell.me

- kbussell
- Kevin Sjöberg, http://kevinsjoberg.com
- Kevin Sumner http://kevinsumner.com
- Kevin Wu http://kevinformatics.com
- Ley Missailidis, http://polymet.is
- Loa https://github.com/loa
- Lowe Schmidt http://loweschmidt.se
- Marcello Barnaba http://sindro.me/
- Marcin Kulik - http://ku1ik.com
- Marius Rieder http://www.durchmesser.ch/
- Mark Allen [@bytemeorg](http://twitter.com/bytemeorg)
- Mark Percival http://markpercival.us
- Mark Sprevak
- Martin Ansdell-Smith https://ansdell.net/
- Martin Kopischke http://martin.kopischke.net
- masukomi http://masukomi.org
- Matt Greer -- http://mattgreer.org
- MetaCosm | #Vim on Freenode
- Michael Ulm www.mulm.at
- Mikael Jansson http://mikaelj.se
- Mikkel Høgh http://mikkel.hoegh.org/
- Ming Liu http://ming.codes
- Holger Peters http://www.holger-peters.de
- Alexander Myshov http://myshov.com/
- Darren Cheng http://sanguinerane.com/
- Felix Rath www.0fx.eu
- Hong Xu http://www.topbug.net
- Slava Kim https://www.meteor.com
- Nate Soares (So8res) http://so8r.es
- neektza http://pltconfusion.com

- Niclas Nilsson, http://niclasnilsson.se
- Nikolay Bachiyski http://extrapolate.me/
- Nikush Patel http://nikush.co.uk
- Norman Köhring https://koehr.in
- Osamu Komagata
- Øystein E. Krog https://github.com/oysteinkrog
- Pablo Brasero Moreno http://pablobm.com
- Peter Souter http://www.petersouter.co.uk
- Phil Levchenko (http://philslair.com)
- Pierre Douyon https://github.com/pgdouyon
- Prateek Rungta
- Przemysław Pietrzkiewicz, http://ppi.sh
- Rasmus Mattsson http://zzm.se
- Reed Esau https://github.com/reedes
- Richard Harding https://bmark.us
- Rob Warner http://grailbox.com
- Runar Balstad Jensen, http://stdout.no/
- Ryan Paul http://seg.phault.net
- Saad Malik - https://github.com/saamalik
- Saggi Malachi - http://sagg.im
- Scott Wessels (http://www.usgn.net)
- Sean DuBois (Sean-Der) http://siobud.com
- Sebastian Vetter (http://www.roadside-developer.com)
- Simen Endsjø, http://simendsjo.me
- Stanley Chan (Happy-Dude)
- Stefan Penner
- Steve Vermeulen https://github.com/svermeulen
- Steven Myint (https://github.com/myint)
- Tae Sandoval Murgan <taecilla.github.io>
- The Kompanee http://thekompanee.com

- Thomas Cannon <http://thomascannon.net>
- Tim Oxley http://campjs.com/
- Timo Schmiade
- Timothy Dahlin
- Tiziano Santoro
- Toban Wiebe http://tobanwiebe.com
- Tobias Sjösten http://vvv.tobiassjosten.net/
- Tom Cammann http://github.com/takac
- Val Markovic http://val.markovic.io/
- Víðir Valberg Guðmundsson http://rabotnik.coop
- Weiyi Lou http://parlseygardens.net
- Wesley Moore http://wezm.net/
- Woody Peterson, http://www.bitsofsignal.com
- Zach Kelling, http://zeekay.io

Your name in BACKERS.md.

- @a13xb
- @d1eg0
- @derkha
- @georgebashi
- @tmcw
- @xpayn
- Adam Baxter (voltagex)
- Adam Miller (maxamillion)
- Adam Sunderland/iterion
- Adon Metcalfe (adonm)
- Adrian Duyzer
- Aidan Stein
- ak47
- Aketzu
- Alessandro Gangemi

- Alessandro Pagnin
- Alessandro Proscia
- Alex Genco (alexgenco)
- Alex Heeton
- Alex Roper
- Alex Soto (apsoto@gmail.com)
- Alexander Flatter
- Alexander Jeurissen
- Alexander Shabalin
- Alexander Woerndle
- Alexandru Keszeg
- Allan Hudgins
- Ami Chayun
- Amit Beka
- Anders Fuzz
- Andre Azzolini
- Andre da Palma
- André Santos/andrefs
- Andrea Michi
- Andreas Heider
- Andreas Lappe
- andrej ocenas / aocenas
- Andrew Arminio @5thWall
- Andrew DeMaria ~ muff1nman
- Andrew Fallows
- Andrew Fecheyr (@andruby)
- Andrew Hartford
- Andrey Tarantsov
- Andrey Yurovsky
- Andy Dirnberger

- Andy Gayton
- Andy Lindeman
- Andy Webster
- AngelLeliel
- Anton Egorov / satyrius
- Anton Shemerey
- Arnaud Bétrémieux
- Arne Ehrlich
- Arseny Zarechnev
- Arvind Deshpande
- Astro Jetson Jr
- Attila Maczak
- Avétis KAZARIAN (avetisk)
- BarryKay
- bbinet
- BenBergman
- Bengt Lüers
- Benjamin Bryant
- Bèr "berkes" Kessels
- Bernd Homuth
- Bheesham Persaud
- Bilal Quadri
- Bjorn Tipling
- Bojan Zivanovic
- Boris Searles
- Braden Bruington
- Brandon Liu/thenovices
- Brandon Mathis - @imathis
- Brett Ritter
- Brian Shaver (shakerlxxv)

- Bryan Grohman
- Bryn Edwards
- Calin Iorgulescu
- calind
- Cameron Wood / cewood
- Chris Lopes
- Chris Lord
- Chris Porter
- Chris Sims (@jcsims)
- Christian Anton / fibbs
- Christian Delahousse cdelahousse
- Christian Lange
- Christian Wellenbrock
- Christophe Badoit
- Christopher Lübbemeier
- Christopher Mullins
- Chungha Kim
- ckromero
- Claas-Thido Pfaff
- Claes Mogren
- Clayton Drazner
- claytron
- Colin Cooper
- corytheboyd
- Cristián Alberto Arenas Ulloa / NiñoScript
- Cyril LEPAGNOT/cyrill62
- D3 Designs
- DAddYE (github: DAddYE)
- Dan Webb
- Daniel C. Henning / danielsdesk

- Daniel Egeberg
- Daniel Egger
- Daniel Hodan/czertbytes
- Daniel Latzer
- Daniel Riti
- Daniel Weisser
- Daniele Polencic
- Dave Dixon
- David Collie
- David Galeano
- David Newell
- David Rappo, Bountysource.com
- David Rasch
- David Stensland - terite
- Davit Samvelyan
- Dean Morin
- deffi420
- defrex
- Deraen
- Dewdrops
- Dick Seabrook
- Dominik Ritter
- Don Browne
- Donald Walker
- Donna Martz
- dpc
- dperson
- dvidsilva
- Edan Maor
- Eddy Garcia

- Edward Ash/cazador481
- Elias Tandel Barrionovo
- Elijah Mirecki
- Elliot Winkler (github: mcmire)
- Enric Lluelles
- Eric Fode
- Erik Goldman
- Erik Johnson Github: terminalmage Twitter: @terminalmage
- Erik Nordlund
- Ethan Erchinger/erchn
- Evan Tatarka
- Ewoud van Raamsdonk
- fd0
- Federico Galassi
- Félix Archambault
- Ferdinand Salis-Samaden
- Florian Bartels
- Florian Sachs
- foca
- forbesmyester
- Frédéric de Villamil
- Fredrik Rambris
- Dapeng Li (luislee818)
- Janko Marohnić (janko-m)
- Jun Wu
- Maurice van der Pot (Griffon26)
- Ory Band (oryband)
- Garrett Dawson
- Gaveen Prabhasara
- George Nikolopoulos

- github username "alandyer"
- Giuseppe Rota
- gkalas
- Gökhan Kocak
- goodgravy
- Göran Gustafsson (GLEG)
- Graham Jans
- Greg Marcil
- Gregory Byrd/GByrd
- Gustavo Barron / Cicloid
- Gyuri Horák
- Harald Hvaal
- Harm Aarts
- Harry Glaser
- Håvard Grimelid
- hencjo
- Henning Hasemann
- Henry "Ingvij" Kupty
- henry4k
- Heryandi
- Hontvári József Levente
- IKEDA, Yoshifumi
- Ilia Choly
- iliis
- Ilya Katz
- Indrek Juhkam / indrekj
- irwand
- Islam Sharabash / ibash
- Ivar Vasara
- Jack Stouffer

- Jacqueline Leykam
- Jakob Landbo
- James Herdman
- James Magness
- James Murty / jmurty
- James Nguyen
- James Orr
- James Pickard
- James Seward
- Jamie Hillman
- Jan Larres
- Jan Weitz
- Janko Luin
- Jannik Nielsen / bitkid
- Jared Tyler Miller
- Jason Imison
- Jason Long / @jasonlong
- Jason Stillwell/dragonfax
- Jasu
- javaguirre
- Jedidiah Hurt
- Jeff Schornick
- Jen Patrick
- Jens Hausherr
- Jess Brown / Jess
- Jesse Cooke/@jc00ke
- Jessica Frazelle/ jfrazelle
- Jesus Alvarez
- Jez Allan
- Jimmie Elvenmark

- Jiří Koten
- JJ Murre
- joe di castro
- Joe Malin/633squadron
- Joel Meyer
- Joey Trapp
- Johan Klokkhammer Helsing
- John Cant
- John K. Paul
- John Krueger
- John Netherdrake
- John Schmidt / schmee
- John Whitley/@jwhitley
- Jon Anhold
- Jon Gosting
- Jonas Rollsby
- Jonathan Buschmann
- Jonathan Gibert
- Jonathan Jacobs
- Jonathan Kinred
- Jorge Gallegos (kad)
- Joris Morger
- Josemar Luedke
- Joshua Brookover
- Joshua Travis
- Jouko Karvonen
- jstemmer
- Juan Campa
- Juan Hernández Babón / jhbabon
- juanolon

- Justin Carter Moy
- Justin Force
- Kamil Slowikowski
- Karl Ove Hufthammer
- Kazuo Teramoto
- Kevin Goslar
- Kevin Hamer
- Kevin Watters
- Kevin Zimmerman / zim44
- kryft
- Krzysztof Adamski
- Lachlan Brad Easton
- Lance Ulmer
- Larry Riedel (Larry@Riedel.org)
- Lasse Immonen (lasse.immonen@gmail.com)
- Léo Unbekandt / Soulou
- Leszek Swirski
- Levi Brown
- lhl
- linduxed
- Lluis Satorre Gonzalez
- lobachevsky
- Louis Pilfold
- Lucas Stephanou/lucasts
- Lucas Werkmeister
- Luke Sampson
- Luke van der Hoeven / @plukevdh
- Maciej Litwiniuk (@mlitwiniuk)
- Mads Navntoft Noe
- Maik Derstappen / MrTango

- Manuel Salvadores
- Marcus Huntemann/mapclyps
- Marcus Martin/nymaen
- Markus Dobler
- Mat Moore
- Mate Nagy
- Mathias Fußenegger
- Matt Moretti
- Matthew Machuga
- Matthew Sunderland
- Matthias Bilger
- Matthias Lehmann
- Maximilian Gerlach
- Meryn Stol
- Michael "manveru" Fellinger
- Michael "beefsack" Alexander
- Michael Iles
- Michael Irwin/mdi
- Michael Kebe
- Michael Lerch
- Michael R. Jones
- Michael Sanders/msanders
- Michael Schall / schallm
- Michail Denchev
- Michał Bartoszkiewicz
- Mickaël FORTUNATO
- Miguel Santesmases
- Mihir Pendse
- Mike Bissett/paran01d
- Mike Sergio

- Mike Wadsten
- mikedillion
- Mikkel Oscar Lyderik Larsen
- Miles Edland / edlandm
- Miles Frain
- Mirko Grönroos
- Moritz Lotze
- Morten Nygaard Åsnes
- MrException
- Julian Churchill (julianchurchill)
- Nakul Dhotre
- Corey Farwell (frewsxcv)
- Tarrant Rollins (tarrant)
- Naseer Ahmed
- Nate Jones
- Nate Soares (So8res)
- Nate Straz
- Neil Kirsopp
- Nelson Chen/nelsonjchen
- Nicholas Firth-McCoy (nfm)
- Nick Sloan
- Nickolas Means
- Nicolai Ruckel
- Niko Kivelä
- nop
- Nuno Frade
- Olaf Heinemann
- Ole Reifschneider
- Oliver Caldwell
- Olivier Lechevalier

- orangain
- orestis
- pablasso
- Pascal Hartig
- Patrick Berkeley
- Patrick Bihan-Faou, TeamBox SARL
- Patrick Stapfer
- Paul R Alexander
- Pavel Afanasyev
- Pawel Grzech
- Paweł Nowak
- Pedro Rodriguez T
- Per Modin
- Peter Edge
- Peter Fern
- Philipe Farias/philipefarias
- Philipp Millar
- Piotr Mitoraj
- Prithvi Prabhu (prithvi)
- Qingping Hou / houqp
- Quasar Jarosz
- queria
- Rachid Zarouali / Xinity
- Radek Szymczyszyn
- Randy Topliffe / Taar
- Ranko Radonić
- Raphael Henrique Ribas
- Ravi Joseph Pinto
- rekab
- Renato Zannon

- Richard "RichiH" Hartmann
- Richard Mihalovič
- Richard Sotarsh Clark
- Rio Kierkels
- rob allen
- Rob Bevan
- Rob Miller @robmil
- Rob O'Dwyer
- Rodolfo Borges (aka bart9h)
- Rodrigo Souto
- Roland Szabo
- Romans Malinovskis
- Roshan George
- Rostepher
- runar
- Russ Adams / rustushki
- Ryan Phillips
- Ryan Pierce/ fission1110
- Ryan Schmitt
- Ryan Tarpine
- Sadik Gokhan Caglar
- Sam Stokes
- Sanpi
- Santiago Suarez Ordoñez
- saxman7000
- Scott D'Aquila
- Scott Mathson
- Sean Copenhaver (@copenhas)
- Sean McCleary
- Sean M. Collins (sc68cal)

- Sean Tegtmeyer
- seanbell
- Sebastian Geiger
- Sergey Alexandrov
- Shane Delmore
- shawndumas
- shazow
- Silvan Jegen
- Simon Ratner (simonratner)
- Song Gao
- Stanimir Angelov Mladenov
- Stanislav B. - zbstof
- Stefan Gojan (hoschi)
- Stefan Hoffmann / stefan991
- Stefan Wienert
- Steffen Rademacker
- Stephen Oney/soney
- Steve "ryz" Haßenpflug
- Steven Daniel Webb
- Steven Sloss
- Sven Opitz
- Sverre Rabbelier
- Szymon Ząbkiewicz
- Taichi Nakamura
- Takayuki Soga
- Teasp00n
- Terin Stock - @terinjokes
- Thai Pangsakulyanont (dtinth)
- Thameera Senanayaka
- Theo Belaire

- Thomas Ferris Nicolaisen
- Thomas Hirsch
- Thomas Maffia/Szerated
- Thomas Pani
- Thomas Thune Hansen
- Thorbjørn Lindeijer
- Thunder Hu
- Thys Meintjes
- Tim Anema
- Tim Matthews
- Timo Kröger
- Tolstyak
- Tom Panning
- Tom Wadley
- Tony Kemp
- tonyo112
- Tori "TicViking" Hamblin
- treppo
- Truman Ellis
- Tuomas Vähänen
- Tyler Briles/MrBri
- Tyler Brock
- Tyler Mandry
- Ulf Fischer/ulf
- Vangelis Tsoumenis
- Vítězslav Homolka
- Vladimir Chernis
- Vojtech Jasny
- vvakame
- Vytis Valentinavičius

- Wade Berrier
- warrenv
- whateverhuis
- Will Binns-Smith
- Xavier Antoviaque (xavier@antoviaque.org)
- xsb
- Yann Villessuzanne
- yko
- Youta Harada(cohalz)
- Yrjö Peussa
- Yusuke Kanda(@uskanda)
- Yuval Langer
- zerowidth
- Ziling Zhao
- Zsolt Botykai

ANONYMOUS SUPPORTERS

There were also 307 other people who didn't claim any level of reward but contributed to the fundraiser. Thank you all for the support!

Nvim

The items listed below are deprecated: they will be removed in the future. They should not be used in new scripts, and old scripts should be updated.

Deprecated features

DEPRECATED IN 0.13 deprecated-0.13

API

- nvim_win_set_height() Use nvim_win_resize() instead.
- nvim_win_set_width() Use nvim_win_resize() instead.
- The "buffer" key accepted and/or returned by these functions, was renamed to "buf":
  - nvim_create_autocmd()
  - nvim_get_autocmds()
  - nvim_exec_autocmds()
  - nvim_clear_autocmds()

EVENTS

- BufModifiedSet Use OptionSet with pattern "modified" instead.

HIGHLIGHTS

- vim.hl.on_yank() Use vim.hl.hl_op() instead.

LUA

- vim.F.if_nil() Renamed to vim.nonnil()
- vim.F.ok_or_nil()
- vim.F.npcall() Renamed to vim.npcall()
- vim.F.nil_wrap() Use vim.npcall() instead

TREESITTER

- vim.treesitter.get_captures_at_pos() Use vim.treesitter.get_captures() with `(buf, { row, col })` instead.

UI

- v:echospace Its purpose to avoid a "Press ENTER" prompt no longer applies with ui2.

LSP

- vim.lsp.util.character_offset() Use to vim.str_utfindex() instead

VIMSCRIPT

- ctxget(), ctxpop(), ctxpush(), ctxset(), ctxsize(). Use nvim_get_context() and nvim_load_context() instead.

DEPRECATED IN 0.12 deprecated-0.12

API

- nvim_set_decoration_provider `on_line` Use `on_range` instead.

DIAGNOSTICS

- "float" in vim.diagnostic.JumpOpts. Use `on_jump` instead.
- "float" in vim.diagnostic.Opts.Jump. Use `on_jump` instead.

HIGHLIGHTS

- :ownsyntax w:current_syntax Use 'winhighlight' instead.

LSP

- vim.lsp.client_is_stopped() Use vim.lsp.get_client_by_id() instead.
- vim.lsp.util.stylize_markdown() Use vim.treesitter.start() with `vim.wo.conceallevel = 2`.
- vim.lsp.log.should_log() Use vim.lsp.log.set_format_func() instead and return `nil` to omit entries from the logfile.
- vim.lsp.semantic_tokens.start() Use `vim.lsp.semantic_tokens.enable(true)` instead
- vim.lsp.semantic_tokens.stop() Use `vim.lsp.semantic_tokens.enable(false)` instead
- vim.lsp.set_log_level() Use `vim.lsp.log.set_level()` instead
- vim.lsp.get_log_path() Use `vim.lsp.log.get_filename()` instead
- vim.lsp.get_buffers_by_client_id() Use `vim.lsp.get_client_by_id(id).attached_buffers` instead
- vim.lsp.stop_client() Use Client:stop() instead
- vim.lsp.codelens.refresh() Use `vim.lsp.codelens.enable(true)` instead
- vim.lsp.codelens.clear() Use `vim.lsp.codelens.enable(false)` instead

- vim.lsp.codelens.display()
- vim.lsp.codelens.save()
- vim.lsp.codelens.on_codelens()

LUA

- vim.diff() Renamed to vim.text.diff()
- "buffer" in `vim.keymap.set.Opts` and `vim.keymap.del.Opts` Renamed to "buf"

UI

- hit-enter hit-enter-prompt With ui2, the legacy "Press ENTER" press-enter prompt is never triggered.

DEPRECATED IN 0.11 deprecated-0.11

API

- nvim_notify() Use nvim_echo() or `nvim_exec_lua("vim.notify(...)", ...)` instead.
- nvim_subscribe() Plugins must maintain their own "multicast" channels list.
- nvim_unsubscribe() Plugins must maintain their own "multicast" channels list.
- nvim_out_write() Use nvim_echo().
- nvim_err_write() Use nvim_echo() with `{err=true}`.
- nvim_err_writeln() Use nvim_echo() with `{err=true}`.
- nvim_buf_add_highlight() Use vim.hl.range() or nvim_buf_set_extmark()

DIAGNOSTICS

- vim.diagnostic.goto_next() Use vim.diagnostic.jump() with `{count=1, float=true}` instead.
- vim.diagnostic.goto_prev() Use vim.diagnostic.jump() with `{count=-1, float=true}` instead.
- vim.diagnostic.get_next_pos() Use the "lnum" and "col" fields from the return value of vim.diagnostic.get_next() instead.
- vim.diagnostic.get_prev_pos() Use the "lnum" and "col" fields from the return value of vim.diagnostic.get_prev() instead.
- The "win_id" parameter used by various functions is deprecated in favor of "winid" winid
- vim.diagnostic.JumpOpts renamed its "cursor_position" field to "pos".

HIGHLIGHTS

- TermCursorNC Unfocused terminal windows do not have a cursor.

LSP

- vim.lsp.util.jump_to_location Use vim.lsp.util.show_document() with `{focus=true}` instead.
- vim.lsp.buf.execute_command Use Client:exec_cmd() instead.
- vim.lsp.buf.completion Use vim.lsp.completion.get() instead.
- vim.lsp.buf_request_all The `error` key has been renamed to `err` inside the result parameter of the handler.
- vim.lsp.with() Pass configuration to equivalent functions in `vim.lsp.buf.*`.
- vim.lsp.handlers Does not support client-to-server response handlers. Only supports server-to-client requests/notification handlers.
- vim.lsp.handlers.signature_help() Use vim.lsp.buf.signature_help() instead.
- `client.request()` Use Client:request() instead.
- `client.request_sync()` Use Client:request_sync() instead.
- `client.notify()` Use Client:notify() instead.
- `client.cancel_request()` Use Client:cancel_request() instead.
- `client.stop()` Use Client:stop() instead.
- `client.is_stopped()` Use Client:is_stopped() instead.
- `client.supports_method()` Use Client:supports_method() instead.
- `client.on_attach()` Use Client:on_attach() instead.
- `vim.lsp.start_client()` Use vim.lsp.start() instead.

LUA

- vim.region() Use getregionpos() instead.
- vim.highlight Renamed to vim.hl.
- vim.validate(opts: table) Use form 1. See vim.validate().

VIMSCRIPT

- termopen() Use jobstart() with `{term: v:true}`.

DEPRECATED IN 0.10 deprecated-0.10

API

- nvim_buf_get_option() Use nvim_get_option_value() instead.
- nvim_buf_set_option() Use nvim_set_option_value() instead.
- nvim_call_atomic() Use nvim_exec_lua() instead.
- nvim_get_option() Use nvim_get_option_value() instead.
- nvim_set_option() Use nvim_set_option_value() instead.
- nvim_win_get_option() Use nvim_get_option_value() instead.
- nvim_win_set_option() Use nvim_set_option_value() instead.

CHECKHEALTH

- health#report_error vim.health.report_error() Use vim.health.error() instead.
- health#report_info vim.health.report_info() Use vim.health.info() instead.
- health#report_ok vim.health.report_ok() Use vim.health.ok() instead.
- health#report_start vim.health.report_start() Use vim.health.start() instead.
- health#report_warn vim.health.report_warn() Use vim.health.warn() instead.

DIAGNOSTICS

- Configuring diagnostic-signs using :sign-define or sign_define(). Use the "signs" key of vim.diagnostic.config() instead.
- vim.diagnostic functions:
  - vim.diagnostic.disable() Use vim.diagnostic.enable()
  - vim.diagnostic.is_disabled() Use vim.diagnostic.is_enabled()
  - Legacy signature: `vim.diagnostic.enable(buf:number, namespace:number)`

LSP

- vim.lsp.util.get_progress_messages() Use vim.lsp.status() instead.
- vim.lsp.get_active_clients() Use vim.lsp.get_clients() instead.
- vim.lsp.for_each_buffer_client() Use vim.lsp.get_clients() instead.
- vim.lsp.util.trim_empty_lines() Use vim.split() with `trimempty` instead.
- vim.lsp.util.try_trim_markdown_code_blocks()
- vim.lsp.util.set_lines()
- vim.lsp.util.extract_completion_items()
- vim.lsp.util.parse_snippet()
- vim.lsp.util.text_document_completion_list_to_complete_items()
- vim.lsp.util.lookup_section() Use vim.tbl_get() instead:

  ```
  local keys = vim.split(section, '.', { plain = true })
  local  vim.tbl_get(table, unpack(keys))
  ```

  LUA
- vim.loop Use vim.uv instead.
- vim.tbl_add_reverse_lookup()
- vim.tbl_flatten() Use IterArray:flatten() instead.
- vim.tbl_islist() Use vim.islist() instead.

OPTIONS

- The "term_background" UI option ui-ext-options is deprecated and no longer populated. Background color detection is now performed in Lua by the Nvim core, not the TUI.

TREESITTER

- LanguageTree:for_each_child() Use LanguageTree:children() (non-recursive) instead.

DEPRECATED IN 0.9 deprecated-0.9

API

- nvim_get_hl_by_name() Use nvim_get_hl() instead.
- nvim_get_hl_by_id() Use nvim_get_hl() instead.

TREESITTER

- vim.treesitter.language.require_language() Use vim.treesitter.language.add() instead.
- vim.treesitter.get_node_at_pos() Use vim.treesitter.get_node() instead.
- vim.treesitter.get_node_at_cursor() Use vim.treesitter.get_node() and TSNode:type() instead.
- The following top level Treesitter functions have been moved:
  - vim.treesitter.inspect_language() -> vim.treesitter.language.inspect()
  - vim.treesitter.get_query_files() -> vim.treesitter.query.get_files()
  - vim.treesitter.set_query() -> vim.treesitter.query.set()
  - vim.treesitter.query.set_query() -> vim.treesitter.query.set()
  - vim.treesitter.get_query() -> vim.treesitter.query.get()
  - vim.treesitter.query.get_query() -> vim.treesitter.query.get()
  - vim.treesitter.parse_query() -> vim.treesitter.query.parse()
  - vim.treesitter.query.parse_query() -> vim.treesitter.query.parse()
  - vim.treesitter.add_predicate() -> vim.treesitter.query.add_predicate()
  - vim.treesitter.add_directive() -> vim.treesitter.query.add_directive()
  - vim.treesitter.list_predicates() -> vim.treesitter.query.list_predicates()
  - vim.treesitter.list_directives() -> vim.treesitter.query.list_directives()
  - vim.treesitter.query.get_range() -> vim.treesitter.get_range()
  - vim.treesitter.query.get_node_text() -> vim.treesitter.get_node_text()

LUA

- nvim_exec() Use nvim_exec2() instead.

- vim.pretty_print() Use vim.print() instead.

DEPRECATED IN 0.8 OR EARLIER

API

- nvim_buf_clear_highlight() Use nvim_buf_clear_namespace() instead.
- nvim_buf_set_virtual_text() Use nvim_buf_set_extmark() instead.
- nvim_command_output() Use nvim_exec2() instead.
- nvim_execute_lua() Use nvim_exec_lua() instead.
- nvim_get_option_info() Use nvim_get_option_info2() instead.

COMMANDS

- :rv :rviminfo **Deprecated** alias to :rshada command.
- :wv :wviminfo **Deprecated** alias to :wshada command.

ENVIRONMENT VARIABLES

- $NVIM_LISTEN_ADDRESS
  - **Deprecated** way to:
    - set the server name (use --listen or serverstart() instead)
    - get the server name (use v:servername instead)
    - detect a parent Nvim (use $NVIM instead)
  - Ignored if --listen is given.
  - Unset at startup (after v:servername is initialized). Can be explicitly provided to terminal and jobstart() by the "env" option. Example:

```
call jobstart(['foo'], { 'env': { 'NVIM_LISTEN_ADDRESS': v:servername  } })
```

EVENTS

- BufCreate Use BufAdd instead.
- EncodingChanged Never fired; 'encoding' is always "utf-8".
- FileEncoding Never fired; equivalent to EncodingChanged.
- GUIEnter Never fired; use UIEnter instead.
- GUIFailed Never fired.

KEYCODES

- <MouseDown> Use <ScrollWheelUp> instead.
- <MouseUp> Use <ScrollWheelDown> instead.

HIGHLIGHTS

- hl-VertSplit Use hl-WinSeparator instead.

LSP DIAGNOSTICS

For each of the functions below, use the corresponding function in vim.diagnostic instead (unless otherwise noted). For example, use vim.diagnostic.get() instead of vim.lsp.diagnostic.get().

- vim.lsp.diagnostic.clear() Use vim.diagnostic.hide() instead.
- vim.lsp.diagnostic.disable() Use vim.diagnostic.enable() instead.
- vim.lsp.diagnostic.display() Use vim.diagnostic.show() instead.
- vim.lsp.diagnostic.enable()
- vim.lsp.diagnostic.get()
- vim.lsp.diagnostic.get_all() Use vim.diagnostic.get() instead.
- vim.lsp.diagnostic.get_count() Use vim.diagnostic.count() instead.
- vim.lsp.diagnostic.get_line_diagnostics() Use vim.diagnostic.get() instead.
- vim.lsp.diagnostic.get_next()
- vim.lsp.diagnostic.get_next_pos()
- vim.lsp.diagnostic.get_prev()
- vim.lsp.diagnostic.get_prev_pos()
- vim.lsp.diagnostic.get_virtual_text_chunks_for_line() No replacement. Use options provided by vim.diagnostic.config() to customize virtual text.
- vim.lsp.diagnostic.goto_next()
- vim.lsp.diagnostic.goto_prev()
- vim.lsp.diagnostic.redraw() Use vim.diagnostic.show() instead.
- vim.lsp.diagnostic.save() Use vim.diagnostic.set() instead.
- vim.lsp.diagnostic.set_loclist() Use vim.diagnostic.setloclist() instead.
- vim.lsp.diagnostic.set_qflist() Use vim.diagnostic.setqflist() instead.
- vim.lsp.diagnostic.show_line_diagnostics() Use vim.diagnostic.open_float() instead.
- vim.lsp.diagnostic.show_position_diagnostics() Use vim.diagnostic.open_float() instead.

The following are deprecated without replacement. These functions are moved internally and are no longer exposed as part of the API. Instead, use vim.diagnostic.config() and vim.diagnostic.show().

- vim.lsp.diagnostic.set_signs()
- vim.lsp.diagnostic.set_underline()
- vim.lsp.diagnostic.set_virtual_text()

LSP FUNCTIONS

- vim.lsp.buf.server_ready() Use LspAttach instead, depending on your use-case. "Server ready" is not part of the LSP spec, so the Nvim LSP client cannot meaningfully implement it. "Ready" is ambiguous because:
  - Language servers may finish analyzing the workspace, but edits can always re-trigger analysis/builds.
  - Language servers can serve some requests even while processing changes.
- vim.lsp.buf.range_code_action() Use vim.lsp.buf.code_action() with the `range` parameter.
- vim.lsp.util.diagnostics_to_items() Use vim.diagnostic.toqflist() instead.
- vim.lsp.util.set_qflist() Use setqflist() instead.
- vim.lsp.util.set_loclist() Use setloclist() instead.
- vim.lsp.buf_get_clients() Use vim.lsp.get_clients() with `{buffer=bufnr}` instead.
- vim.lsp.buf.formatting() Use vim.lsp.buf.format() with `{async=true}` instead.
- vim.lsp.buf.formatting_sync() Use vim.lsp.buf.format() with `{async=false}` instead.
- vim.lsp.buf.range_formatting() Use vim.lsp.formatexpr() or vim.lsp.buf.format() instead.

LUA

- vim.register_keystroke_callback() Use vim.on_key() instead.

NORMAL COMMANDS

-]f [f Same as "gf".

OPTIONS

- cpo-< :menu-<special> :menu-special :map-<special> :map-special <> notation is always enabled.
- 'fe' 'fenc'+'enc' before Vim 6.0; no longer used.
- 'gdefault' boolean (default off) Enables the :substitute "/g" flag by default. See complex-change. command 'gdefault' on 'gdefault' off ~ :s/// subst. all subst. one :s///g subst. one subst. all :s///gg subst. all subst. one **WARNING:** Setting this breaks plugins, because "/g" will have the opposite effect. :s_g
- 'highlight' 'hl' Names of builtin highlight-groups cannot be changed.
- 'isprint' 'isp' Printability of ASCII or Unicode chars cannot be changed.
- 'langnoremap' **Deprecated** alias to 'nolangremap'.
- 'magic' 'nomagic' boolean (default on) Changes the special characters in search patterns. **WARNING:** Disabling this breaks plugins, because most patterns assume it's on. See also /\M.
- 'paste' 'nopaste' Just Paste It.™ The 'paste' option is obsolete: paste is handled automatically when you paste text using your terminal's or GUI's paste feature (`CTRL-SHIFT-v`, CMD-v (macOS), middle-click, …). Enables "paste mode":
  - Disables mappings in Insert, Cmdline mode.
  - Disables abbreviations.
  - Resets 'autoindent' 'expandtab' 'revins' 'ruler' 'showmatch' 'smartindent' 'smarttab' 'softtabstop' 'textwidth' 'wrapmargin'.
  - Treats 'formatoptions' as empty.
  - Disables the effect of these options:
    - 'cindent'
    - 'indentexpr'
    - 'lisp'
- 'sessionoptions' Flags "unix", "slash" are ignored and always enabled.
- 'vi'
- 'viewoptions' Flags "unix", "slash" are ignored and always enabled.
- 'viminfo' **Deprecated** alias to 'shada' option.
- 'viminfofile' **Deprecated** alias to 'shadafile' option.

UI EXTENSIONS

- ui-wildmenu Use ui-cmdline with ui-popupmenu instead. Enabled by the `ext_wildmenu` ui-option. Emits these events:
  - `["wildmenu_show", items]`
  - `["wildmenu_select", selected]`
  - `["wildmenu_hide"]`
- term_background Unused. The terminal background color is now detected by the Nvim core directly instead of the TUI.

VIMSCRIPT

- <sfile> Use <script> or <stack> instead.
- b:terminal_job_pid Use `jobpid(&channel)` instead.
- b:terminal_job_id Use `&channel` instead. To access in non-current buffer:
  - Lua: `vim.bo[bufnr].channel`
  - Vimscript: `getbufvar(bufnr, '&channel')`
- buffer_exists() Obsolete name for bufexists().
- buffer_name() Obsolete name for bufname().
- buffer_number() Obsolete name for bufnr().
- file_readable() Obsolete name for filereadable().
- highlight_exists() Obsolete name for hlexists().
- highlightID() Obsolete name for hlID().
- inputdialog() Use input() instead.
- jobclose() Obsolete name for chanclose()
- jobsend() Obsolete name for chansend()
- last_buffer_nr() Obsolete name for bufnr("$").
- rpcstart() Use jobstart() with `{'rpc': v:true}` instead.

- rpcstop() Use jobstop() instead to stop any job, or `chanclose(id, "rpc")` to close RPC communication without stopping the job. Use chanclose(id) to close any socket.

Development of Nvim dev

This reference describes design constraints and guidelines, for developing Nvim applications or Nvim itself. See dev-arch for discussion of Nvim's architecture and internal concepts.

Nvim is free and open source. Everybody is encouraged to contribute. https://github.com/neovim/neovim/blob/master/CONTRIBUTING.md

Design goals design-goals

Most important things come first (roughly). Some items conflict; this is intentional. A balance must be found.

NVIM IS... IMPROVED design-improved

The Neo bits of Nvim should make it a better Vim, without becoming a completely different editor.

- In matters of taste, prefer Vim/Unix tradition. If there is no relevant Vim/Unix tradition, consider the "common case".
- There is no limit to the features that can be added. Select new features based on (1) what users ask for, (2) how much effort it takes to implement and (3) someone actually implementing it.
- Backwards compatibility is a feature. The RPC API in particular should never break.

NVIM IS... WELL DOCUMENTED design-documented

- A feature that isn't documented is a useless feature. A patch for a new feature must include the documentation.
- Documentation should be comprehensive and understandable. Use examples.
- Don't make the text unnecessarily long. Less documentation means that an item is easier to find.

NVIM IS... FAST AND SMALL design-speed-size

Keep Nvim small and fast. This directly affects versatility and usability.

- Computers are becoming faster and bigger each year. Vim can grow too, but no faster than computers are growing. Keep Vim usable on older systems.
- Many users start Vim from a shell very often. Startup time must be short.
- Commands must work efficiently. The time they consume must be as small as possible. Useful commands may take longer.
- Don't forget that some people use Vim over a slow connection. Minimize the communication overhead.
- Vim is a component among other components. Don't turn it into a massive application, but have it work well together with other programs ("composability").

NVIM IS... MAINTAINABLE design-maintain

- The source code should not become a mess. It should be reliable code.
- Use comments in a useful way! Quoting the function name and argument names is NOT useful. Do explain what they are for.
- Porting to another platform should be made easy, without having to change too much platform-independent code.
- Use the object-oriented spirit: Put data and code together. Minimize the knowledge spread to other parts of the code.

NVIM IS... NOT design-not

Nvim is not an operating system; instead it should be composed with other tools or hosted as a component. Marvim once said: "Unlike Emacs, Nvim does not include the kitchen sink... but it's good for plumbing."

Developer guidelines dev-guidelines

Providers dev-provider

A primary goal of Nvim is to allow extension of the editor without special knowledge in the core. Some core functions are delegated to "providers" implemented as external scripts.

Examples:

1. In the Vim source code, clipboard logic accounts for more than 1k lines of C source code (ui.c), to perform two tasks that are now accomplished with shell commands such as xclip or pbcopy/pbpaste.
2. Python scripting support: Vim has three files dedicated to embedding the Python interpreter: if_python.c, if_python3.c and if_py_both.h. Together these files sum about 9.5k lines of C source code. In contrast, Nvim Python scripting is performed by an external host process implemented in ~2k lines of Python.

The provider framework invokes Vimscript from C. It is composed of two functions in eval.c:

- eval_call_provider({name}, {method}, {arguments}, {discard}): Calls `provider#{name}#Call` with {method} and {arguments}. If {discard} is true, any value returned by the provider will be discarded and empty value will be returned.
- eval_has_provider({name}): Checks the `g:loaded_{name}_provider` variable which must be set to 2 by the provider script to indicate that it is "enabled and working". Called by has() to check if features are available.

For example, the Python provider is implemented by the "autoload/provider/python.vim" script, which sets `g:loaded_python_provider` to 2 only if a valid external Python host is found. Then `has("python")` reflects whether Python support is working.

provider-reload

Sometimes a GUI or other application may want to force a provider to "reload". To reload a provider, undefine its "loaded" flag, then use :runtime to reload it:

```
:unlet g:loaded_clipboard_provider
:runtime autoload/provider/clipboard.vim
```

Documentation

dev-doc

- "Just say it". Avoid mushy, colloquial phrasing in all documentation (docstrings, user manual, website materials, newsletters, ...). Don't mince words. Personality and flavor, used sparingly, are welcome--but in general, optimize for the reader's time and energy: be "precise yet concise".
  - Prefer the active voice: "Foo does X", not "X is done by Foo".
  - "The words you choose are an essential part of the user experience." https://developer.apple.com/design/human-interface-guidelines/writing
  - "...without being overly colloquial or frivolous." https://developers.google.com/style/tone
- Write docstrings (as opposed to inline comments) with present tense ("Gets"), not imperative ("Get"). This tends to reduce ambiguity and improve clarity by describing "What" instead of "How".

```
✅ OK:
/// Gets a highlight definition.
❌ NO:
/// Get a highlight definition.
```

- Avoid starting docstrings with "The" or "A" unless needed to avoid ambiguity. This is a visual aid and reduces noise.

```
✅ OK:
/// @param dirname Path fragment before `pend`
❌ NO:
/// @param dirname The path fragment before `pend`
```

- Vim differences:
  - Do not prefix help tags with "nvim-". Use vim_diff.txt to catalog differences from Vim; no other distinction is necessary.
  - If a Vim feature is removed, delete its help section and move its tag to vim_diff.txt.
- Mention deprecated features in deprecated.txt and delete their old doc.
- Use consistent language.
  - "terminal" in a help tag always means "the embedded terminal emulator", not "the user host terminal".
  - Use "tui-" to prefix help tags related to the host terminal, and "TUI" in prose if possible.
- Rough guidelines on where Lua documentation should end up:
  - Nvim API functions `vim.api.nvim_*` should be in `api.txt`.
  - If the module is big and not relevant to generic and lower-level Lua functionality, then it's a strong candidate for separation. Example: `treesitter.txt`
  - Otherwise, add them to `lua.txt`

DOCUMENTATION FORMAT

For Nvim-owned docs, use the following strict subset of "vimdoc" to ensure the help doc renders nicely in other formats (such as HTML: https://neovim.io/doc/user).

Strict "vimdoc" subset:

- Use lists (like this!) prefixed with "-" or "•", for adjacent lines that you don't want to auto-wrap. Lists are always rendered with "flow" layout (soft-wrapped) instead of preformatted (hard-wrapped) layout common in legacy :help docs. You can also use numbered lists, e.g. "1."
- Separate blocks (paragraphs) of content by a blank line.
- Do not use indentation in random places—that prevents the page from using "flow" layout. If you need a preformatted section, put it in a help-codeblock starting with ">".
- Parameters and fields are documented as `{foo}`.
- Optional parameters and fields are documented as `{foo}?`.

C DOCSTRINGS

Nvim API documentation lives in the source code, as docstrings (doc comments) on the function definitions. The api :help is generated from the docstrings defined in src/nvim/api/*.c.

Docstring format:
- Lines start with `///`
- Special tokens start with `@` followed by the token name: `@note`, `@param`, `@return`
- Markdown is supported.
- Tags are written as `[tag]()`.
- References are written as `[tag]`
- Use "```" for code samples. Code samples can be annotated as `vim` or `lua`

Example: the help for nvim_open_win() is generated from a docstring defined in src/nvim/api/win_config.c like this:

```
/// Opens a new window.
/// ...
///
/// Example (Lua): window-relative float
///
/// ```lua
/// vim.api.nvim_open_win(0, false, {
///   relative='win',
///   row=3,
///   col=3,
///   width=12,
///   height=3,
/// })
/// ```
///
/// @param buffer Buffer to display
/// @param enter  Enter the window
/// @param config Map defining the window configuration. Keys:
///   - relative: Sets the window layout, relative to:
///      - "editor" The global editor grid.
///      - "win"    Window given by the `win` field.
///      - "cursor" Cursor position in current window.
/// ...
/// @param[out] err Error details, if any
///
/// @return Window handle, or 0 on error
```

LUA DOCSTRINGS

dev-lua-doc

Lua documentation lives in the source code, as docstrings on the function definitions. The lua-vim :help is generated from the docstrings.

Docstring format:

- Use LuaCATS annotations: https://luals.github.io/wiki/annotations/
- Markdown is supported.
- Tags are written as `[tag]()`.
- References are written as `[tag]`
- Use "```" for code samples. Code samples can be annotated as `vim` or `lua`
- Use `@since <api-level>` to note the api-level when the function became "stable". If `<api-level>` is greater than the current stable release (or 0), it is marked as "experimental".
  - See scripts/util.lua for the mapping of api-level to Nvim version.
- Use `@deprecated` to mark deprecated functions.
- For callable module tables, document `__call` in `setmetatable(M, { ... })`.
- Use `@nodoc` to prevent documentation generation.
- Use `@private` for private functions.
  - Naming a function with an underscore prefix ("_foo", not "foo") implies `@nodoc` and `@private`.
- Use `@internal` for non-public functions shared across Nvim modules, or `@class (internal)` for shared internal types. Both hide generated docs.
- Use `@inlinedoc` to inline `@class` blocks into `@param` blocks. E.g.

```
--- Object with fields:
--- @class myOpts
--- @inlinedoc
---
--- Documentation for some field
--- @field somefield? integer

--- @param opts? myOpts
function foo(opts)
end
```

Will be rendered as:

```
foo({opts})

    Parameters:
      - {opts}? (table) Object with the fields:
                - {somefield}? (integer) Documentation
                  for some field
```

- Files declared as `@meta` are only used for typing and documentation (similar to "*.d.ts" typescript files).

Example: the help for vim.paste() is generated from a docstring decorating vim.paste in runtime/lua/vim/_core/editor.lua like this:

```
--- Paste handler, invoked by |nvim_paste()| when a conforming UI
--- (such as the |TUI|) pastes text into the editor.
---
--- Example: To remove ANSI color codes when pasting:
---
--- ```lua
--- vim.paste = (function()
---   local overridden = vim.paste
---   ...
--- end)()
--- ```
---
--- @since 12
--- @see |paste|
---
--- @param lines  ...
--- @param phase  ...
--- @returns false if client should cancel the paste.
```

EXX ERROR CODES

dev-error-codes

To choose a new "EXX" error code (e.g. E5555), just print a message with a new EXX number:

```
emsg(_("E996: Invalid thing"));
```

To see existing error codes:

```
:exe 'help' | let g:errtags=sort(map(taglist('E\d\d\d\?\d\?'), 'v:val.name'), 'N') | new | put =g:errtags
```

The `linterrcodes` build target checks that all "EXX" codes are unique and have help tags. It runs in CI, but you can also run it locally:

```
make linterrcodes
```

Stdlib

dev-lua

GUIDANCE

- Lua names generally should conform to dev-naming.
- Keep the core Lua modules lua-stdlib simple. Avoid elaborate OOP or pseudo-OOP designs. Plugin authors just want functions to call, not a big, fancy inheritance hierarchy.
- Avoid requiring or returning special objects in the Nvim stdlib. Plain tables or values are easier to serialize, easier to construct from literals, easier to inspect and print, and inherently compatible with all Lua plugins. (This guideline doesn't apply to opaque, non-data objects like `vim.cmd`.)
- stdlib functions should follow these common patterns:
  - Return lua-result-or-message (`any|nil,nil|string`) to communicate failure, or choose from dev-error-patterns when appropriate.
  - Accept iterable instead of only table.
    - **Note:** in some cases iterable doesn't make sense, e.g. spair() sorts the input by definition, so there is no reason for it to accept an iterable, because the input needs to be "reified"; it can't operate on a "stream".
  - Return an iterable (generator) instead of table, if possible.
  - Mimic the pairs() or ipairs() interface if the function is intended for use in a for-in loop.

ERRORS

dev-error-patterns

To communicate failure to a consumer, choose from these patterns (in order of preference):

1. `retval, errmsg`
   - When failure is normal, or when it is practical for the consumer to continue (fallback) in some other way. See lua-result-or-message.
2. optional result, no errormsg
   - Special case of 1. When there is only a single case of "doesn't exist" (e.g. cache lookup, dict lookup).
3. `error("no luck")`
   - For invalid state ("must not happen"), when failure is exceptional, or at a low level where the consumers are unlikely to handle it in a meaningful way. Advantage is that propagation happens for free and it's harder to accidentally swallow errors. (E.g. using `uv_handle/pipe:write()` without checking return values is common.)
4. `on_error` callback
   - For async and "visitors" traversing a graph, where many errors may be collected while work continues.
5. `nvim_echo` (consider `opts.silent` for async/visitors)
   - High-level / application-level messages. End-user invokes these directly.

API

dev-api

Where applicable, this section applies to both Lua stdlib and RPC API.

GUIDANCE

- When adding an API, check the following:
  - What precedents did you draw from? How does your solution compare to them?
  - Does your new API allow future expansion? How? Or why not?
  - Is the new API similar to existing APIs? Do we need to deprecate the old ones?
  - Did you cross-reference related concepts in the docs?
- Avoid "mutually exclusive" parameters (via constraints or limitations, if necessary). For example nvim_create_autocmd() has mutually exclusive "callback" and "command" args; but the "command" arg could be eliminated by simply not supporting Vimscript function names,

and treating a string "callback" arg as an Ex command (which can call Vimscript functions). The "buffer" arg could also be eliminated by treating a number "pattern" as a buffer number.
- Avoid functions that depend on cursor position, current buffer, etc. Instead the function should take a position parameter, buffer parameter, etc.

WHERE THINGS GO

- API (libnvim/RPC): exposes low-level internals, or fundamental things (such as `nvim_exec_lua()`) needed by clients or C consumers.
- Lua stdlib = high-level functionality that builds on top of the API.

See also dev-layout.

API PATTERNS

dev-api-patterns

- When accepting a buffer id, etc., 0 means "current buffer", nil means "all buffers". Likewise for window id, tabpage id, etc.
  - Examples: vim.lsp.codelens.clear() vim.diagnostic.enable()
- Any function signature that accepts a callback (example: table.foreach()) should place it as the LAST parameter (after opts), if possible (or ALWAYS for continuations——functions called exactly once).
  - Improves readability by placing the less "noisy" arguments near the start.
  - Consistent with luv.
  - Useful for future async lib which transforms functions of the form `function(<args>, cb(<ret>)) => function(<args>) -> <ret>`.
  - Example:

```
-- ✅ OK:
filter(…, opts, fn)
-- ❌ NO:
filter(…, fn, opts)
```

- If the module or topic wants to support configuration:
  - Prefer a new 'option' (or family of related options, or "xxoptions" style such as 'jumpoptions'). Options get these benefits "for free":
    - Completion: `:set a<tab>`, `:lua vim.o.a<tab>`
    - OptionSet event
    - Scopes: global/buf/win/tab (can start as global, then "graduate" later)
    - UI: `:verbose set`, `:options`, etc.
  - Alternatively if the editor options system absolutey doesn't fit: Expose a `config(opts)` interface (example: vim.diagnostic.config()). If `opts` is omitted (or `nil`) it returns the current configuration.
- "Enable" ("toggle") interface and behavior:
  - `enable(…, nil)` and `enable(…, {buf=nil})` are synonyms and control the "global" enablement of a feature.
    - `is_enabled(nil)` and `is_enabled({buf=nil})`, likewise, query the global state of the feature.
  - `enable(…, {buf: number})` sets a buffer-local "enable" flag.
    - `is_enabled({buf: number})`, likewise, queries the buffer-local state of the feature.
  - See vim.lsp.inlay_hint.enable() and vim.lsp.inlay_hint.is_enabled() for a reference implementation of these "best practices".
  - **NOTE:** open questions: https://github.com/neovim/neovim/issues/28603
- Transformation functions should also have "filter" functionality (when appropriate): when the function returns a nil value it excludes (filters out) its input, else the transformed value is used.
  - Example: See the format() field of vim.diagnostic.Opts.Float.
- Do not add "foreach" functions (e.g. LanguageTree:for_each_tree()). Instead return an iterable, which is more flexible and composable:

```
vim.iter(parser.tree_iter()):each(function()
  ...
end)
```

NAMING

dev-naming

Naming is exceedingly important: the name of a thing is the primary interface for uses it, discusses it, searches for it, shares it... Consistent naming in the stdlib, API, and UI helps both users and developers discover and intuitively understand related concepts ("families"), and reduces cognitive burden. Discoverability encourages code re-use and likewise avoids redundant, overlapping mechanisms, which reduces code surface-area, and thereby minimizes bugs...

In general, look for precedent when choosing a name, that is, look at existing (non-deprecated) functions. In particular, see below...

dev-name-common

Use existing common `{verb}` names (actions) if possible:
- add: Appends or inserts into a collection
- attach: Listens to something to get events from it (TODO: rename to "on"?)
- call: Calls a function
- callback continuation: Function parameter (NOT a field) that returns the result of an async function. Use the "on_..." naming-convention for all other callbacks and event handlers.
- cancel: Cancels or dismisses an event or interaction, typically user-initiated and without error. (Compare "abort", which cancels and signals error/failure.)
- clear: Clears state but does not destroy the container
- close: Releases/destroys a resource.
- create: Creates a new (non-trivial) thing (TODO: rename to "def"?)
- del: Deletes a thing (or group of things)
- detach: Dispose attached listener (TODO: rename to "un"?)
- enable: Enables/disables functionality. Signature should be `enable(enable?:boolean, filter?:table)`.
- eval: Evaluates an expression
- exec: Executes code, may return a result
- fmt: Formats
- get: Gets things. Two variants (overloads):

    1. `get<T>(id: int): T` returns one item.
    2. `get<T>(filter: dict): T[]` returns a list.
- has: Checks for the presence of an item, feature, etc.
- inspect: Presents a high-level, often interactive, view
- is_enabled: Checks if functionality is enabled.
- on_…: Handles events or async results, or registers such a handler. dev-name-events
- open: Opens something (a buffer, window, …)
- parse: Parses something into a structured form
- request: Calls a remote (network, RPC) operation.
- send: Writes data or a message to a channel.
- set: Sets a thing (or group of things)
- start: Spin up a long-lived process. Prefer "enable" except when "start" is obviously more appropriate.
- stop: Inverse of "start". Teardown a long-lived process.
- try_{verb}: Best-effort operation, failure returns null or error obj

Do NOT use these deprecated verbs:
- contains: Prefer "has".
- destroy: Prefer "close".
- disable: Prefer `enable(enable: boolean)`.
- exit: Prefer "cancel" (or "stop" if appropriate).
- is_disabled: Prefer `!is_enabled()`.
- list: Redundant with "get"
- notify: Redundant with "print", "echo"
- show: Redundant with "print", "echo"
- toggle: Prefer `enable(not is_enabled())`.

Use consistent names for `{topic}` in API functions: buffer is called "buf" everywhere, not "buffer" in some places and "buf" in others.
- buf: Buffer
- chan: channel
- cmd: Command
- cmdline: Command-line UI or input
- dir: Directory
- fn: Function
- hl: Highlight
- pos: Position
- proc: System process
- tabpage: Tabpage
- win: Window

Do NOT use these deprecated nouns:
- buffer Use "buf" instead
- command Use "cmd" instead
- window Use "win" instead

continuation

A "continuation" is implemented as a callback function that returns the result of an async function and is called exactly once. Often accepts `err` as the first parameter, to avoid erroring on the main thread. Example:

```
foo(…, callback: fun(err, …))
```

Use the name `callback` only for a continuation. All other callbacks and event handlers should be named with the `on_…` convention. dev-name-events

dev-namespace-name

Use namespace names like `nvim.foo.bar`:

```
vim.api.nvim_create_namespace('nvim.lsp.codelens')
```

dev-augroup-name

Use autocommand group names like `nvim.foo.bar`:

```
vim.api.nvim_create_augroup('nvim.treesitter.dev')
```

dev-name-events

Use the "on_" prefix to name event-handling callbacks and also the interface for "registering" such handlers (on_key). The dual nature is acceptable to avoid a confused collection of naming conventions for these related concepts.

Editor events (autocommands) are historically named like:

```
{Noun}{Event}
```

RPC API events use this format:

```
nvim_{noun}_{event}_event (example: nvim_buf_changedtick_event)
```

RPC API NAMING

dev-api-name

Use this format to name new RPC API functions:

```
nvim_{topic}_{verb}_{arbitrary-qualifiers}
```

Do not add new nvim_buf/nvim_win/nvim_tabpage APIs, unless you are certain the concept will NEVER be applied to more than one "scope". That is, `{topic}` should be the TOPIC ("ns", "extmark", "option", …) that acts on the scope(s) (buf/win/tabpage/global), it should NOT be the

scope. Instead the scope should be a parameter (typically manifest as mutually-exclusive buf/win/... flags like nvim_get_option_value(), or less commonly as a `scope: string` field like nvim_get_option_info2()).

- Example: `nvim_get_keymap('v')` operates in a global context (first parameter is not a Buffer). The "get" verb indicates that it gets anything matching the given filter parameter. A "list" verb is unnecessary because `nvim_get_keymap('')` (empty filter) returns all items.
- Example: `nvim_buf_del_mark` acts on a `Buffer` object (the first parameter) and uses the "del" `{verb}`.

The RPC API doesn't have "modules" so its functions are grouped by a naming convention. Prefer the `nvim_{topic}_{verb}_…` convention where possible.

Example:

```
nvim_ns_add(
  ns_id: int,
  filter: {
    handle: integer (buf/win/tabpage id)
    scope: "global" | "win" | "buf" | "tabpage"
  }
): { ok: boolean }

nvim_ns_get(
  ns_id: int,
  filter: {
    handle: integer (buf/win/tabpage id)
    scope: "global" | "win" | "buf" | "tabpage"
  }
): { ids: int[] }

nvim_ns_del(
  ns_id: int,
  filter: {
    handle: integer (buf/win/tabpage id)
    scope: "global" | "win" | "buf" | "tabpage"
  }
): { ok: boolean }
```

Anti-Example:

Creating separate `nvim_xx`, `nvim_buf_xx`, `nvim_win_xx`, and `nvim_tabpage_xx`, functions all for the same xx topic, requires 4x the amount of documentation, tests, boilerplate, and interfaces, which users must comprehend, maintainers must maintain, etc. Thus the following is NOT recommended (compare these 12(!) functions to the above 3 functions):

```
nvim_add_ns(…)
nvim_buf_add_ns(…)
nvim_win_add_ns(…)
nvim_tabpage_add_ns(…)
nvim_del_ns(…)
nvim_buf_del_ns(…)
nvim_win_del_ns(…)
nvim_tabpage_del_ns(…)
nvim_get_ns(…)
nvim_buf_get_ns(…)
nvim_win_get_ns(…)
nvim_tabpage_get_ns(…)
```

Interface patterns

dev-interface-patterns

EX COMMANDS

dev-excmd

Prefer "flat" Ex commands (`:foobar`) instead of "subcommands" (`:foo bar`). except when the benefits below outweigh the costs.

Pros of subcommands:

- Useful when subcommands are discovered _at runtime_ (e.g. `:checkhealth foo`). Plugins can extend the dispatcher by injecting subcommands.
- `:Foo<tab>` completes only the Foo subcommands, whereas completing a flat name matches every command sharing the prefix (e.g. `:FooBar` and `:FootballBar`).
- Completion can vary dynamically: e.g. `:lsp <tab>` can hide `stop` if no client is running. (But this can be an antifeature, because it makes the behavior less predictable.)

Cons of subcommands:

- Less harmonious with fuzzy pickers. Pickers can easily list `:foobar`, whereas enumerating subcommands requires special logic. And in a picker, `:foo bar` is often two "steps".
- Adds a step when tab-completing.
- Awkward to support `[!]`, ranges, `[count]`, etc., per-subcommand.
- Implementation cost: a dispatcher with custom completion.

API client

dev-api-client

api-client

API clients wrap the Nvim API to provide idiomatic "SDKs" for their respective platforms (see jargon). You can build a new API client for your favorite platform or programming language.

List of API clients: https://github.com/neovim/neovim/wiki/Related-projects#api-clients

node-client pynvim These clients can be considered the "reference implementation" for API clients:

- https://github.com/neovim/node-client
- https://github.com/neovim/pynvim

STANDARD FEATURES

- API clients exist to hide msgpack-rpc details. The wrappers can be automatically generated by reading the api-metadata from Nvim. api-mapping
- Clients should call nvim_set_client_info() after connecting, so users and plugins can detect the client by handling the ChanInfo event. This avoids the need for special variables or other client hints.
- Clients should handle nvim_error_event notifications, which will be sent if an async request to nvim was rejected or caused an error.

PACKAGE NAMING

API client packages should NOT be named something ambiguous like "neovim" or "python-client". Use "nvim" as a prefix/suffix to some other identifier following ecosystem conventions.

For example, Python packages tend to have "py" in the name, so "pynvim" is a good name: it's idiomatic and unambiguous. If the package is named "neovim", it confuses users, and complicates documentation and discussions.

Examples of API-client package names:
- ✅ OK: nvim-racket
- ✅ OK: pynvim
- ❌ NO: python-client
- ❌ NO: neovim_

API CLIENT IMPLEMENTATION

- Separate the transport layer from the rest of the library. rpc-connecting
- Use a MessagePack library that implements at least version 5 of the MessagePack spec, which supports the BIN and EXT types used by Nvim.
- Use a single-threaded event loop library/pattern.
- Use a fiber/coroutine library for the language being used for implementing a client. These greatly simplify concurrency and allow the library to expose a blocking API on top of a non-blocking event loop without the complexity that comes with preemptive multitasking.
- Don't assume anything about the order of responses to RPC requests.
- Clients should expect requests, which must be handled immediately because Nvim is blocked while waiting for the client response.
- Clients should expect notifications, but these can be handled "ASAP" (rather than immediately) because they won't block Nvim.
- For C/C++ projects, consider libmpack instead of the msgpack.org library. https://github.com/libmpack/libmpack/ libmpack is small (no dependencies, can inline into your C/C++ project) and efficient (no allocations). It also implements msgpack-RPC, the protocol required by Nvim. https://github.com/msgpack-rpc/msgpack-rpc

External UI **dev-ui**

External UIs should be aware of the api-contract. In particular, future versions of Nvim may add new items to existing events. The API is strongly backwards-compatible, but clients must not break if new (optional) fields are added to existing events.

STANDARD FEATURES

External UIs are expected to implement these common features:

- Call nvim_set_client_info() after connecting, so users and plugins can detect the UI by handling the ChanInfo event. This avoids the need for special variables and UI-specific config files (gvimrc, macvimrc, ...).
- Cursor style (shape, color) should conform to the 'guicursor' properties delivered with the mode_info_set UI event.
- Send the ALT/META ("Option" on macOS) key as a <M- chord.
- Send the "super" key (Windows key, Apple key) as a <D- chord.
- Avoid mappings that conflict with the Nvim keymap-space; GUIs have many new chords (`<C-,>` `<C-Enter>` `<C-S-x>` `<D-x>`) and patterns ("shift shift") that do not potentially conflict with Nvim defaults, plugins, etc.
- Consider the "option_set" ui-global event as a hint for other GUI behaviors. Various UI-related options ('guifont', 'ambiwidth', ...) are published in this event. See also "mouse_on", "mouse_off".
- UIs generally should NOT set $NVIM_APPNAME (unless explicitly requested by the user).
- Support the text decorations/attributes given by ui-event-hl_attr_define. The "url" attr should be presented as a clickable hyperlink.
- Handle the "restart" UI event so that :restart works.
- Detect capslock and show an indicator if capslock is active.

MULTIGRID UI **dev-ui-multigrid**

- A multigrid UI should display floating windows using one of the following methods, using the `win_float_pos` ui-multigrid event. Different methods can be selected for each window as needed.
  1. Let Nvim determine the position and z-order of the windows. This can be achieved by using the `compindex`, `screen_row`, and `screen_col` information from the `win_float_pos` UI event. To allow Nvim to automatically determine the grid, the UI should call `nvim_input_mouse` with 0 as the `grid` parameter.
  2. Use the `anchor`, `anchor_grid`, `anchor_row`, `anchor_col`, and `zindex` as references for positioning the windows. If windows are outside the screen, it is the UI's responsibility to reposition and/or resize them. Since Nvim lacks information about actual window placement in this case, the UI must call `nvim_input_mouse` with the actual grid id, factoring in `mouse_enabled`. **Note:** Some API functions may return unexpected results for these windows due to the missing information.
- For external windows, the grid id should also be passed to `nvim_input_mouse`.

How to develop Nvim, explanation of modules and subsystems

Module-specific details are documented at the top of each module (`terminal.c`, `undo.c`, …). The top of each major module has (or should have) an overview in a comment at the top of its file.

The purpose of this document is to give:

1. an overview of how it all fits together
2. how-to guides for common tasks such as:
   - (TODO) deprecating public functions
   - (TODO) adding a new public (API) function or (UI) event

Project layout: where things go

dev-layout

C CODE

- Nvim C code lives in `src/nvim/`.
  - This includes third-party components which we have forked and fully "own", meaning we no longer track the upstream. In particular: `vterm/` and `tui/termkey/`.
- Vendored third-party components live in `src/*`.
  - `src/tee/` and `src/xxd/` have their own build files. They are shipped with the Windows artifact to ensure those tools exists there.
  - Other components like `src/cjson/` and `src/mpack/` are included with Nvim itself. These vendored sources are synced from upstream, we do not "own" them.

See also "MAINTAIN.md".

Generated C files use these filename conventions to hint their purpose:
- `*.c`, `*.generated.c` - full C files, with all includes, etc.
- `*.c.h` - parametrized C files, contain all necessary includes, but require defining macros before actually using. Example: `typval_encode.c.h`
- `*.h` - full headers, with all includes. Does not apply to `*.generated.h`.
- `*.h.generated.h` - exported functions' declarations.
- `*.c.generated.h` - static functions' declarations.

LUA CODE

Lua code lives in one of four places:

1. Plugins! Not everything needs to live on `vim.*`. Plugins are the correct model for non-essential functionality which (1) has "autoload" behavior, (2) the user may disable or replace with a third-party plugin, (3) is primarily user-facing, not an API.
   - "Opt-out" plugins (activated on startup): `runtime/plugin/`
   - "Opt-in" plugins (activated via `:packadd`): `runtime/pack/dist/opt/`
   - **Note:** Update standard-plugin-list.
   - **Note:** New plugins should place Lua modules in the shared "nvim" namespace: `require('nvim.foo')`, not `require('foo')`.
   - Examples:
     - ✅ spellfile `runtime/lua/nvim/spellfile.lua`
     - ✅ tutor
   - Don't use these legacy plugins as examples (their modules should live in `lua/nvim/x.lua`, not `lua/x.lua`):
     - ❌ editorconfig.lua
     - ❌ man.lua
     - ❌ nvim.difftool
     - ❌ nvim.tohtml
     - ❌ nvim.undotree
2. Stdlib: `runtime/lua/vim/`. Modules on `vim.*`, lazy-loaded (as opposed to "Core" modules, see below).
   - Examples: `vim.treesitter`, `vim.lpeg`.
3. Core: `runtime/lua/vim/_core/`. Internal-only, compiled-in modules. May directly interact with Nvim internals. Only available in main thread. See dev-lua-builtin below.
4. "Shared" core: `runtime/lua/vim/_core/shared.lua`. Pure Lua functions which always are available. Used in the test runner, and worker threads/processes launched from Nvim.

The top-level `vim.` namespace is for fundamental Lua and editor features. Use submodules (`vim.foo`) for everything else (but avoid excessive "nesting"), or plugins (see above).

Compatibility with Vim's `if_lua` is explicitly a non-goal.

dev-lua-builtin

Lua modules that are necessary even if VIMRUNTIME is invalid (generally, data structures and essential utilities), must be compiled into the `nvim` binary. There are two ways to do that:
- if they are private, put the code file in `runtime/lua/vim/_core/`
- if they are public, choose a new filepath in `runtime/lua/vim/` and add it to VIM_MODULE_FILE in src/nvim/CMakeLists.txt.

Data structures

- StringBuilder
- kvec or garray.c for dynamic lists / vectors (use StringBuilder for strings)

Use `kvec.h` for most lists. When you absolutely need a linked list, use `src/nvim/lib/queue_defs.h` which defines an "intrusive" linked list.

Buffer text is stored as a tree of line segments, defined in `src/nvim/memline.c`. The central idea is found in `ml_find_line`.

Many of the editor concepts are defined as Lua data files:

- Events (autocmds): src/nvim/auevents.lua
- Ex (cmdline) commands: src/nvim/ex_cmds.lua

- Options: src/nvim/options.lua
- Vimscript functions: src/nvim/eval.lua
  - Functions can be implemented in C (`f_foo` in eval/funcs.c), or in Lua (set `func_lua` in eval.lua, implement in _core/vimfn.lua). Lua impl is preferred, for performance and maintainability.
    - VimScript callers go through `lua_wrapper` (typval <=> Object conversion).
    - Lua callers (`vim.fn.foo()`) take a fast path in `nlua_call()` that calls the Lua function directly, avoiding any conversion.
- v: variables: src/nvim/vvars.lua

Events

dev-events

The events historically called "autocmds", referred to here as "editor events" or simply "events", are high-level events for use by plugins, user config, and the Nvim editor. (There is an unrelated, low-level concept defined by the `event/defs.h#Event` struct, which is just a bag of data passed along the internal event-loop.)

Where possible, new editor events should be implemented using `aucmd_defer()` (and where possible, old events migrate to this), so they are processed in a predictable manner, which avoids crashes. See dev-new-event.

UI events

dev-ui-events

The long-term vision is that UI events are just another type of "editor event" (formerly known as "autocmds"). There is no real reason that we have separate types of user-facing or plugin-facing events. Events are events. Their "transport" is irrelevant and any event should be possible to emit over any transport (editor or RPC).

Meanwhile the current situation is that UI events are a particular RPC event packaged in a generic `redraw` notification. They also can be listened to in-process via vim.ui_attach().

UI events are deferred to UIs, which implies a deepcopy of the UI event data.

The source files most directly involved with UI events are:
1. `src/nvim/ui.*`: calls handler functions of registered UI structs (independent from msgpack-rpc)
2. `src/nvim/api/ui.*`: forwards messages over msgpack-rpc to remote UIs.

UI events are defined in `src/nvim/api/ui_events.in.h` , this file is not compiled directly, rather it parsed by `src/gen/gen_api_ui_events.lua` which autogenerates wrapper functions used by the source files above. It also generates metadata accessible as `api_info().ui_events`.

See commit d3a8e9217f39c59dd7762bd22a76b8bd03ca85ff for an example of adding a new UI event. Remember to bump NVIM_API_LEVEL if it wasn't already during this development cycle.

Other references:
- msgpack-rpc
- ui
- https://github.com/neovim/neovim/pull/3246
- https://github.com/neovim/neovim/pull/18375
- https://github.com/neovim/neovim/pull/21605

State

dev-state

"Editor state" is represented by these (sometimes overlapping) concepts:

- shada (`src/nvim/shada.c`): user data: registers, marks, ... Msgpack format.
- sessions/views (`src/nvim/ex_session.c`): windows/layout, per-window cursor/options, CWD, ... Vimscript format: commands, not merged data. Overlaps shada for the buffer list and global variables.
- context (`src/nvim/context.c`): nvim_get_context(). Interfaces shada and other state; also in-memory per-cursor state (`kCtxVisual`, `kCtxMarks`, ...) for multicursor. Needs more thought.
- scoped execution ("temporary state"): `ctx_switch()`, `vim._with()`, nvim_win_call(), nvim_buf_call(), `cmdmod_T`. Temporary switches, not snapshots. The C mechanisms overlap with each other; `vim._with()` is the unified interface.
- reentrancy protection: save/restore around nested execution. `InsState`, `RedoState`.
  - TODO: ad hoc pairs remain for typeahead (`tasave_T`), v:event, view state, cmdline, with inconsistent nesting semantics (stack vs refcount vs single-shot).

LONG-TERM VISION(?): shada is the authority on all user-data state, sessions on layout state? New state kinds extend the shada schema instead of adding a format or an in-memory sidecar. Eliminate context.c?

LUA FUNCTIONS ACROSS PROCESSES

The test harness transfers Lua closures between Nvim instances (`test/functional/testnvim/exec_lua.lua`). This works ONLY between same-build instances (LuaJIT bytecode not portable across versions/builds) and only for trusted peers. Upvalues are limited to msgpack-able values (not function/userdata; tables lose metatables and identity).

Filesystem

dev-filesystem

https://github.com/neovim/neovim/issues/36112 Path handling, especially separator chars ("/" vs "\"), is designed as follows: "/" separators are used every except at the "edges", i.e. the separator chars are coverted just-in-time, and only when absolutely needed. This is similar to how UTF-8 encoding is used internally for all text, but buffers can read/write various encodings.

P.S. Windows handles "/" separators just fine, except in rare cases. It'll be fine, don't worry. Just use "/", for god's sake!

API

dev-api-fast

API functions and Vimscript "eval" functions may be marked as api-fast which means they are safe to call in Lua callbacks and other scenarios. A function CANNOT be marked as "fast" if it could trigger `os_breakcheck()`, which may "yield" the current execution and start a new execution of code not expecting this:

- accidentally recursing into a function not expecting this.
- changing (global) state without restoring it before returning to the "yielded" callsite.

In practice, this means any code that could trigger `os_breakcheck()` cannot be "fast". For example, commit 3940c435e405 fixed such a bug with `nvim__get_runtime` by explicitly disallowing `os_breakcheck()` via the `EW_NOBREAK` flag.

Common examples of non-fast code: regexp matching, wildcard expansion, expression evaluation.

Internal practices and patterns

dev-internals-howto

Add a new excmd or vimfn ("f_xx") function

dev-new-excmd **dev-new-vimfn**

To implement an Ex cmd or f_xx function, choose from one of three ways (in order of preference):

1. Full Lua: the implementation lives in entirely in Lua. This has a performance benefit: the Vimscript <=> Lua bridge is skipped when the `vim.fn.xx()` function is called from Lua. Examples:
   - `f_hostname`
2. Partial Lua: the C implementation calls `nlua_call_typval` or `nlua_call_excmd`. (Or in rare cases: `nlua_exec` / `NLUA_EXEC_STATIC`). Examples:
   - `ex_log`
   - `ex_lsp`
   - `f_serverlist`
3. Legacy way: implement the function entirely in C. Examples:
   - `f_chansend`
   - `f_flatten`
   - `f_searchpos`

Add a new event (autocmd)

dev-new-event

To add a new editor event (autocmd):

1. Register the event name in `src/nvim/auevents.lua`. If the event is Nvim-only (not in Vim), also update the `nvim_specific` table.
2. Fire the event, by one of these approaches:
   - From C, use `aucmd_defer()` to fire deferred (safe, predictable). See `do_markset_autocmd` for example.
   - From Lua, use `vim.api.nvim_exec_autocmds()`.
3. Define the event-data type (if any) in `runtime/lua/vim/_meta/events.lua`.
4. Document the event in `runtime/doc/autocmd.txt`.

The event-loop

event-loop

The internal, low-level, libuv event-loop (luv-event-loop) is used to schedule arbitrary work in a predictable way. One such obvious use-case for scheduling is deferred editor-events (autocmds). Another example is job-control.

ASYNC EVENT SUPPORT

One of the features Nvim added is the support for handling arbitrary asynchronous events, which can include:

- RPC requests
- job control callbacks
- timers

Nvim implements this functionality by entering another event loop while waiting for characters, so instead of:

```
def state_enter(on_state, data):
  do
    key = readkey()           # Read a key from the user
  while on_state(data, key)   # Invoke callback for the current state
```

the Nvim program loop is more like:

```
def state_enter(on_state, data):
  do
    event = read_next_event() # Read an event from the OS
  while on_state(data, event) # Invoke callback for current state
```

where event is something the operating system delivers to us, including (but not limited to) user input. The `read_next_event()` part is internally implemented by libuv, the platform layer used by Nvim.

Since Nvim inherited its code from Vim, the states are not prepared to receive "arbitrary events", so we use a special key to represent those (When a state receives an "arbitrary event", it normally doesn't do anything other than update the screen).

MAIN LOOP

The Loop structure (which describes `main_loop`) abstracts multiple queues into one loop:

```
uv_loop_t uv;
MultiQueue *events;
MultiQueue *thread_events;
MultiQueue *fast_events;
```

`loop_poll_events` checks `Loop.uv` and `Loop.fast_events` whenever Nvim is idle, and also at `os_breakcheck` intervals.

MultiQueue is cool because you can attach throw-away "child queues" trivially. For example `do_os_system()` does this (for every spawned process!) to automatically route events onto the `main_loop`:

```
Process *proc = &uvproc.process;
MultiQueue *events = multiqueue_new_child(main_loop.events);
proc->events = events;
```

NVIM LIFECYCLE

How Nvim processes input.

Consider a typical Vim-like editing session:

1. Vim displays the welcome screen
2. User types: `:`
3. Vim enters command-line mode
4. User types: `edit README.txt<CR>`
5. Vim opens the file and returns to normal mode
6. User types: `G`
7. Vim navigates to the end of the file
8. User types: `5`
9. Vim enters count-pending mode
10. User types: `d`
11. Vim enters operator-pending mode
12. User types: `w`
13. Vim deletes 5 words
14. User types: `g`
15. Vim enters the "g command mode"
16. User types: `g`
17. Vim goes to the beginning of the file
18. User types: `i`
19. Vim enters insert mode
20. User types: `word<ESC>`
21. Vim inserts "word" at the beginning and returns to normal mode

Note that we split user actions into sequences of inputs that change the state of the editor. While there's no documentation about a "g command mode" (step 16), internally it is implemented similarly to "operator-pending mode".

From this we can see that Vim has the behavior of an input-driven state machine (more specifically, a pushdown automaton since it requires a stack for transitioning back from states). Assuming each state has a callback responsible for handling keys, this pseudocode represents the main program loop:

```
def state_enter(state_callback, data):
  do
    key = readkey()                 # read a key from the user
  while state_callback(data, key)   # invoke the callback for the current state
```

That is, each state is entered by calling `state_enter` and passing a state-specific callback and data. Here is a high-level pseudocode for a program that implements something like the workflow described above:

```
def main()
  state_enter(normal_state, {}):

def normal_state(data, key):
  if key == ':':
    state_enter(command_line_state, {})
  elif key == 'i':
    state_enter(insert_state, {})
  elif key == 'd':
    state_enter(delete_operator_state, {})
  elif key == 'g':
    state_enter(g_command_state, {})
  elif is_number(key):
    state_enter(get_operator_count_state, {'count': key})
  elif key == 'G'
    jump_to_eof()
  return true

def command_line_state(data, key):
  if key == '<cr>':
    if data['input']:
      execute_ex_command(data['input'])
    return false
  elif key == '<esc>'
    return false

  if not data['input']:
    data['input'] = ''

  data['input'] += key
  return true

def delete_operator_state(data, key):
  count = data['count'] or 1
  if key == 'w':
    delete_word(count)
  elif key == '$':
    delete_to_eol(count)
  return false  # return to normal mode

def g_command_state(data, key):
  if key == 'g':
    go_top()
  elif key == 'v':
    reselect()
  return false  # return to normal mode

def get_operator_count_state(data, key):
  if is_number(key):
    data['count'] += key
    return true
  unshift_key(key)  # return key to the input buffer
  state_enter(delete_operator_state, data)
  return false

def insert_state(data, key):
  if key == '<esc>':
    return false  # exit insert mode
  self_insert(key)
  return true
```

The above gives an idea of how Nvim is organized internally. Some states like the `g_command_state` or `get_operator_count_state` do not have a dedicated `state_enter` callback, but are implicitly embedded into other states (this will change later as we continue the refactoring effort). To start reading the actual code, here's the recommended order:

1. `state_enter()` function (state.c). This is the actual program loop, note that a `VimState` structure is used, which contains function pointers for the callback and state data.
2. `main()` function (main.c). After all startup, `normal_enter` is called at the end of function to enter normal mode.
3. `normal_enter()` function (normal.c) is a small wrapper for setting up the NormalState structure and calling `state_enter`.
4. `normal_check()` function (normal.c) is called before each iteration of normal mode.
5. `normal_execute()` function (normal.c) is called when a key is read in normal mode.

The basic structure described for normal mode in 3, 4 and 5 is used for other modes managed by the `state_enter` loop:

- command-line mode: `command_line_{enter,check,execute}()(ex_getln.c)`
- insert mode: `insert_{enter,check,execute}()(insert.c)`
- terminal mode: `terminal_{enter,execute}()(terminal.c)`

IMPORTANT VARIABLES

The current mode is stored in `State`. The values it can have are `MODE_NORMAL`, `MODE_INSERT`, `MODE_CMDLINE`, and a few others.

The current window is `curwin`. The current buffer is `curbuf`. These point to structures with the cursor position in the window, option values, the file name, etc.

All the global variables are declared in `globals.h`.

THE MAIN EVENT-LOOP

The main loop is implemented in state_enter. The basic idea is that Nvim waits for the user to type a character and processes it until another character is needed. Thus there are several places where Nvim waits for a character to be typed. The `vgetc()` function is used for this. It also handles mapping.

Internal logic "stuffs" keys (readahead) only for a consumer on the current call stack: `exec_stuffed()`, or a nested reader it invokes. See "Stuffing" in input.c. (This is simpler and more predictable than Vim, where a command may stuff chars, and let the main loop deal with them at some arbitrary future point, thus requiring global flags checked at the right time, scattered throughout the codebase.)

What we consider the "Nvim event loop" is actually a wrapper around `uv_run` to handle both the `fast_events` queue and possibly (a suitable subset of) deferred events. Therefore "raw" `vim.uv.run()` is often not enough to yield from Lua; instead call `vim.wait(0)` and check its result.

Updating the screen is mostly postponed until a command or a sequence of commands has finished. The work is done by `update_screen()`, which calls `win_update()` for every window, which calls `win_line()` for every line. See the start of drawscreen.c for more explanations.

COMMAND-LINE MODE

When typing a `:`, `normal_cmd()` will call `getcmdline()` to obtain a line with an Ex command. `getcmdline()` calls a loop that will handle each typed character. It returns when hitting <CR> or <Esc> or some other character that ends the command line mode.

EX COMMANDS

Ex commands are handled by the function `do_cmdline()`. It does the generic parsing of the `:` command line and calls `do_one_cmd()` for each separate command. It also takes care of while loops.

`do_one_cmd()` parses the range and generic arguments and puts them in the exarg_t and passes it to the function that handles the command.

The `:` commands are listed in ex_cmds.lua.

NORMAL MODE COMMANDS

The Normal mode commands are handled by the `normal_cmd()` function. It also handles the optional count and an extra character for some commands. These are passed in a `cmdarg_T` to the function that handles the command.

There is a table `nv_cmds` in normal.c which lists the first character of every command. The second entry of each item is the name of the function that handles the command.

INSERT MODE COMMANDS

When doing an `i` or a command, `normal_cmd()` will call the `edit()` function. It contains a loop that waits for the next character and handles it. It returns when leaving Insert mode.

Multicursor

dev-multicursor **dev-cmdatom**

Multicursor is essentially a bunch of extmarks that "cascade" the user action (CmdAtom) driven by the primary cursor.

An "atom" is any user action, as a resolved ("elemental", post-mapping) keysequence, the same material ("redobuf") replayed by dot-repeat (single-repeat). Whereas a macro is a series of meaningless characters whose meaning is context-dependent, assigned at the moment of execution, an atom scopes an input sequence to one, semantic, user action. Examples: "x" is captured as "dl", an insert session as `ciwfoo<Esc>`, a Visual-mode edit as `viweed`.

CONCEPTS

- ATOM: A repeatable unit of user input: the resolved keysequence, plus structured fields (`CmdSpec`). Every user action emits a CmdAtom event.
  - SPAN: Fragment of an Insert or Visual sequence, cascades but does not emit a CmdAtom.
  - FRAME: Scopes one MODE_NORMAL command (Normal, Visual, Select, Op-pending). Non-MODE_NORMAL modes (Insert, Cmdline) are "sessions", not "frames". All emit atoms.
- CASCADE: Replay queued atoms at the "clock edge" at every cursor. Visual edits cascade span-by-span each preceded by "gv"; the non-edit "tail" dry-runs to display the selection.
- COMPOSITE: An atom composed of subatoms (`CmdAtom.atoms`): a mapping/macro, or a Visual sequence.
- INSERT-SESSION: All changes from an insert-mode session, until <Esc>.
  - INSERTION: An `Ins.start`..cursor "undo unit" within a session.
  - INSERT-CASCADE: Replay an insert-session span-by-span as it is typed (each span is packaged as a quasi-session: `i` + keys + <Esc>), via nested `edit()`. Pending literal text is a PREVIEW until its span completes.
- LOSSY: When capture lost part of a composite (payload with no capturing atom, incomplete insert-session): `keys` cannot replay it, `lhs` can.
- PAYLOAD: Keys a command reads interactively while executing: a search/Ex cmdline (`d/pat<CR>`), `f{char}`, a mapping's `getchar()`.
- REPLAY: Execute keys (dot-repeat, or atom).
- VOID: When a pending (Visual) atom's keys were tainted/poisoned (by: mouse, Select mode, a scroll that drags the cursor, …), thus not replayable.

IMPLEMENTATION

Capture happens at the `atom_xx()` hooks.

- `prep_redo()` marks a redoable command (operators, `r`, `J`, `p`, …). Then `atom_cmd_end()` takes the final redobuff as the atom, (including payloads, e.g. `d/pat<CR>`). Prep-exempt commands (yank, `D`, folds) are reconstructed instead, in `atom_capture_op()`.
- `atom_cmd_start()` delimits a command in `normal_execute()`. Everything not covered above is decided here (motions, jumps, scrolls, mouse).

- `atom_ins_start()` delimits an insert session.
- `atom_map_start()`, `atom_macro_start()` delimit a composite.
- `atom_visual_end()` completes the pending Visual atom, `viweex => viweed`. <Esc> discards it; non-replayable commands VOID it.
- `atom_cmdline_set()` gets an invoked cmdline payload (e.g. `/pat<CR>` is a motion atom, `:cnext<CR>` is an "excmd" atom.)

DOT-REPEAT

The redo buffer is the pending change as a `CmdSpec` (`RedoState`): the `["x][count]` prefix is structured fields set by `prep_redo()`, and the command body is bytes appended during execution by the `redo_append_xx()` family (Vim's `AppendToRedobuff*`). Consumers rebuild the keysequence at the boundaries (`redo_keys()`, `start_redo()`, `atom_from_redo()`).

THE CASCADE

Cursors are extmarks with associated `Context` data (`mc_cursors`). The extmark is authoritative: it follows edits, and user code may delete cursors by deleting their extmarks. Cursor-local state is swapped-in on replay, so each cursor reads/writes its own registers, Visual area and change marks.

CmdAtoms queued for cascade (`g_atoms`) are replayed at every cursor on the "clock edge": the completion of a toplevel `normal_execute()`. While a mapping is executing (its keys are still in typebuf) the edge is deferred (subatoms queue/accumulate).

Undo is buffer-global and applied in bulk. Each cascade adds one undo state, and undo/redo itself is not cascaded.

The insert-cascade PREVIEW/COMMIT model (why text is previewed instead of replayed) is documented at `mc_ins_span`. Sessions that cannot insert-cascade (count `3iZ`, Replace mode, blockwise `CTRL-V c`) are captured at <Esc>.
- See `CmdAtomType` and `atom_key_class()` to understand how input is classified to decide whether and how it can be replayed.

RHS VS LHS

Multicursor "cascade" is meaningful only for edits and motions. Example: while you have multicursors placed, you may want to scroll or switch to a window temporarily; it would be a chore if you had to exit multicursor before doing those actions.

Thus, cascade prefers to replay, skip, or partially-apply RHS (the resolved keys) instead of naively replaying LHS (the unresolved, remap-enabled, user input). In general, RHS "degrades gracefully" by doing LESS (and a single "u" reverts it), whereas LHS worst-case is unbounded (it fails by doing MORE; "u" cannot revert its side-effects).

LHS may resolve to arbitrary Excmds/Lua that make sense at the primary, but not at cascade. Whereas RHS classification allows multicursor to skip (or partially apply) a user action that involves unreplayable subatoms. Similar to how insert-completion does not show a menu at every cursor, but it does cascade.

Examples where RHS replay behaves better than LHS:

- Performance:
  - LHS replays the entire callback per-cursor, which can be very slow if it involves treesitter queries, LSP requests, etc.
  - RHS replays only the resolved normal commands.
- Viewport-relative commands (zt/<C-d>/H/L/... in mapping):
  - LHS cascade is nonsense because cursors may be offscreen.
- Jumps / global ops (absolute, not cursor-relative):
  - `nnoremap X Gdd`: RHS replays only dd, per cursor.
  - `nnoremap <Down> ]C`: LHS (follow-mode) moves each cursor to its neighbor.
  - `nnoremap u u`: LHS would undo N times.
- Editor state outside of current buffer:
  - `nnoremap ]x <Cmd>cnext<CR>x`:
    - LHS is nonsense.
    - RHS: `:cnext` runs once at primary; only x cascades.
  - `nnoremap X <Cmd>set wrap!<CR>x`:
    - LHS toggles N times, the result depends on N.
    - RHS does not cascade the <Cmd>.

Examples where LHS replay behaves better (and RHS replay is wrong):

- Conditional mappings, e.g. `nnoremap X <Cmd>if getline('.') =~ 'foo' \| normal! dd \| endif<CR>`.
  - Unblocked by `setrepeat()` API: https://github.com/neovim/neovim/issues/42069

RENDERING

Display is owned by `runtime/lua/vim/_core/mcursor.lua`. The C core maintains the model, as extmarks:

- "nvim.multicursor": the tracking marks (cursor positions).
- "nvim.multicursor.cursor": selection-END display cursors, during Visual mode (each cursor displays at its own selection end). Consumers (UIs) that display cursors use these positions while any exist, instead of the tracking marks.
- "nvim.multicursor.visual": the pending selection ranges.

GUI SUPPORT

A GUI gets a working display for free, via cell highlights. If it wants to draw "real" cursors, it is expected to:
- receive positions via the `win_extmark` UI event: the tracking marks are ui_watched, so visible cursor positions are pushed per-redraw. (Known issue: win_extmark does not give a removal signal; re-query when in doubt.)
- or query the namespaces above with nvim_buf_get_extmarks(); prefer the "nvim.multicursor.cursor" positions while any exist (in-progress Visual selection).
- render selections from "nvim.multicursor.visual", or simply keep their hl-MCursorVisual grid highlights.

Nvim style guide

Style guidelines for developers working on Nvim's source code.

License: CC-By 3.0 https://creativecommons.org/licenses/by/3.0/

Background

One way in which we keep the code base manageable is by enforcing consistency. It is very important that any programmer be able to look at another's code and quickly understand it.

Maintaining a uniform style and following conventions means that we can more easily use "pattern-matching" to infer what various symbols are and what invariants are true about them. Creating common, required idioms and patterns makes code much easier to understand.

In some cases there might be good arguments for changing certain style rules, but we nonetheless keep things as they are in order to preserve consistency.

Header Files

dev-style-header

Header guard

All header files should start with `#pragma once` to prevent multiple inclusion.

In foo/bar.h:

```
#pragma once
```

Headers system

Nvim uses two types of headers. There are "normal" headers and "defs" headers. Typically, each normal header will have a corresponding defs header, e.g. `fileio.h` and `fileio_defs.h`. This distinction is done to minimize recompilation on change. The reason for this is because adding a function or modifying a function's signature happens more frequently than changing a type The goal is to achieve the following:

- All headers (defs and normal) must include only defs headers, system headers, and generated declarations. In other words, headers must not include normal headers.
- Source (.c) files may include all headers, but should only include normal headers if they need symbols and not types.

Use the following guideline to determine what to put where:

Symbols:
- regular function declarations
- `extern` variables (including the EXTERN macro)

Non-symbols:
- macros, i.e. `#define`
- static inline functions with the `FUNC_ATTR_ALWAYS_INLINE` attribute
- typedefs
- structs
- enums
- All symbols must be moved to normal headers.
- Non-symbols used by multiple headers should be moved to defs headers. This is to ensure headers only include defs headers. Conversely, non-symbols used by only a single header should be moved to that header.
- EXCEPTION: if the macro calls a function, then it must be moved to a normal header.

Scoping

dev-style-scope

Local Variables

Place a function's variables in the narrowest scope possible, and initialize variables in the declaration.

C99 allows you to declare variables anywhere in a function. Declare them in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to. In particular, initialization should be used instead of declaration and assignment, e.g.

```
int i;
i = f();      // ❌: initialization separate from declaration.

int j = g();  // ✅: declaration has initialization.
```

Initialization

Multiple declarations can be defined in one line if they aren't initialized, but each initialization should be done on a separate line.

```
int i;
int j;              // ✅
int i, j;           // ✅: multiple declarations, no initialization.
int i = 0;
int j = 0;          // ✅: one initialization per line.

int i = 0, j;       // ❌: multiple declarations with initialization.
int i = 0, j = 0;   // ❌: multiple declarations with initialization.
```

Nvim-Specific Magic

clint

Use `clint.lua` to detect style errors.

`src/clint.lua` is a Lua script that reads a source file and identifies style errors. It is not perfect, and has both false positives and false negatives, but it is still a valuable tool. False positives can be ignored by putting `// NOLINT` at the end of the line.

uncrustify

src/uncrustify.cfg is the authority for expected code formatting, for cases not covered by clint.lua. We remove checks in clint.lua if they are covered by uncrustify rules.

Other C Features

[dev-style-features]

Variable-Length Arrays and alloca()

We do not allow variable-length arrays or `alloca()`.

Variable-length arrays can cause hard to detect stack overflows.

Postincrement and Postdecrement

Use postfix form (`i++`) in statements.

```
for (int i = 0; i < 3; i++) { }
int j = ++i;  // ✅: ++i is used as an expression.

for (int i = 0; i < 3; ++i) { }
++i;  // ❌: ++i is used as a statement.
```

Use of const

Use `const` pointers whenever possible. Avoid `const` on non-pointer parameter definitions.

Where to put the const

Some people favor the form `int const *foo` to `const int *foo` . They argue that this is more readable because it's more consistent: it keeps the rule that `const` always follows the object it's describing. However, this consistency argument doesn't apply in codebases with few deeply-nested pointer expressions since most `const` expressions have only one `const`, and it applies to the underlying value. In such cases, there's no consistency to maintain. Putting the `const` first is arguably more readable, since it follows English in putting the "adjective" (`const`) before the "noun" (`int`).

That said, while we encourage putting `const` first, we do not require it. But be consistent with the code around you!

```
void foo(const char *p, int i);
}

int foo(const int a, const bool b) {
}

int foo(int *const p) {
}
```

Integer Types

Of the built-in integer types only use `char`, `int`, `uint8_t`, `int8_t`, `uint16_t`, `int16_t`, `uint32_t`, `int32_t`, `uint64_t`, `int64_t`, `uintmax_t`, `intmax_t`, `size_t`, `ssize_t`, `uintptr_t`, `intptr_t`, and `ptrdiff_t`.

Use `int` for error codes and local, trivial variables only.

Use care when converting integer types. Integer conversions and promotions can cause non-intuitive behavior. Note that the signedness of `char` is implementation defined.

Public facing types must have fixed width (`uint8_t`, etc.)

There are no convenient `printf` format placeholders for fixed width types. Cast to `uintmax_t` or `intmax_t` if you have to format fixed width integers.

Type unsigned signed `char` `%hhu` `%hhd` `int` n/a `%d` `(u)intmax_t` `%ju` `%jd` `(s)size_t` `%zu` `%zd` `ptrdiff_t` `%tu` `%td`

Booleans

Use `bool` to represent boolean values.

```
int loaded = 1;  // ❌: loaded should have type bool.
```

Conditions

Don't use "yoda-conditions". Use at most one assignment per condition.

```
if (1 == x) {

if (x == 1) {  //use this order

if ((x = f()) && (y = g())) {
```

Function declarations

Every function must not have a separate declaration.

Function declarations are created by the gen_declarations.lua script.

```
static void f(void);

static void f(void)
{
  ...
}
```

General translation unit layout

The definitions of public functions precede the definitions of static functions.

```
<HEADER>

<PUBLIC FUNCTION DEFINITIONS>

<STATIC FUNCTION DEFINITIONS>
```

Integration with declarations generator

Every C file must contain #include of the generated header file.

Include must go after other #includes and typedefs in .c files and after everything else in header files. It is allowed to omit #include in a .c file if .c file does not contain any static functions.

Included file name consists of the .c file name without extension, preceded by the directory name relative to src/nvim. Name of the file containing static functions declarations ends with `.c.generated.h`, `*.h.generated.h` files contain only non-static function declarations.

```
// src/nvim/foo.c file
#include <stddef.h>

typedef int FooType;

#include "foo.c.generated.h"

…


// src/nvim/foo.h file
#pragma once

…

#include "foo.h.generated.h"
```

64-bit Portability

Code should be 64-bit and 32-bit friendly. Bear in mind problems of printing, comparisons, and structure alignment.

- Remember that `sizeof(void *) != sizeof(int)`. Use `intptr_t` if you want a pointer-sized integer.
- You may need to be careful with structure alignments, particularly for structures being stored on disk. Any class/structure with a `int64_t`/`uint64_t` member will by default end up being 8-byte aligned on a 64-bit system. If you have such structures being shared on disk between 32-bit and 64-bit code, you will need to ensure that they are packed the same on both architectures. Most compilers offer a way to alter structure alignment. For gcc, you can use `__attribute__((packed))`. MSVC offers `#pragma pack()` and `__declspec(align())`.
- Use the LL or ULL suffixes as needed to create 64-bit constants. For example:

  ```
  int64_t my_value = 0x123456789LL;
  uint64_t my_mask = 3ULL << 48;
  ```

  sizeof ~

Prefer `sizeof(varname)` to `sizeof(type)`.

Use `sizeof(varname)` when you take the size of a particular variable. `sizeof(varname)` will update appropriately if someone changes the variable type either now or later. You may use `sizeof(type)` for code unrelated to any particular variable, such as code that manages an external or internal data format where a variable of an appropriate C type is not convenient.

```
Struct data;
memset(&data, 0, sizeof(data));

memset(&data, 0, sizeof(Struct));

if (raw_size < sizeof(int)) {
  fprintf(stderr, "compressed record not big enough for count: %ju", raw_size);
  return false;
}
```

Naming

dev-style-naming

The most important consistency rules are those that govern naming. The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring us to search for the declaration of that entity. The pattern-matching engine in our brains relies a great deal on these naming rules.

Naming rules are pretty arbitrary, but we feel that consistency is more important than individual preferences in this area, so regardless of whether you find them sensible or not, the rules are the rules.

General Naming Rules

Function names, variable names, and filenames should be descriptive; eschew abbreviation.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

```
int price_count_reader;    // No abbreviation.
int num_errors;            // "num" is a widespread convention.
int num_dns_connections;   // Most people know what "DNS" stands for.

int n;                     // Meaningless.
int nerr;                  // Ambiguous abbreviation.
int n_comp_conns;          // Ambiguous abbreviation.
int wgc_connections;       // Only your group knows what this stands for.
int pc_reader;             // Lots of things can be abbreviated "pc".
int cstmr_id;              // Deletes internal letters.
```

File Names

Filenames should be all lowercase and can include underscores (_).

Use underscores to separate words. Examples of acceptable file names:

```
my_useful_file.c
getline_fix.c  // ✅: getline refers to the glibc function.
```

C files should end in `.c` and header files should end in `.h`.

Do not use filenames that already exist in `/usr/include`, such as `db.h`.

In general, make your filenames very specific. For example, use `http_server_logs.h` rather than `logs.h`.

Type Names

Typedef-ed structs and enums start with a capital letter and have a capital letter for each new word, with no underscores: `MyExcitingStruct`.

Non-Typedef-ed structs and enums are all lowercase with underscores between words: `struct my_exciting_struct` .

```
struct my_struct {
  ...
};
typedef struct my_struct MyAwesomeStruct;
```

Variable Names

Variable names are all lowercase, with underscores between words. For instance: `my_exciting_local_variable`.

Common Variable names

For example:

```
string table_name;  // ✅: uses underscore.
string tablename;   // ✅: all lowercase.

string tableName;   // ❌: mixed case.
```

Struct Variables

Data members in structs should be named like regular variables.

```
struct url_table_properties {
  string name;
  int num_entries;
}
```

Global Variables

Don't use global variables unless absolutely necessary. Prefix global variables with `g_`.

Constant Names

Use a k followed by mixed case: `kDaysInAWeek`.

All compile-time constants, whether they are declared locally or globally, follow a slightly different naming convention from other variables. Use a k followed by words with uppercase first letters:

```
const int kDaysInAWeek = 7;
```

Function Names

Function names are all lowercase, with underscores between words. For instance: `my_exceptional_function()`. All functions in the same header file should have a common prefix.

In `os_unix.h`:

```
void unix_open(const char *path);
void unix_user_id(void);
```

If your function crashes upon an error, you should append `or_die` to the function name. This only applies to functions which could be used by production code and to errors that are reasonably likely to occur during normal operation.

Enumerator Names

Enumerators should be named like constants: `kEnumName`.

```
enum url_table_errors {
  kOK = 0,
  kErrorOutOfMemory,
  kErrorMalformedInput,
};
```

Macro Names

They're like this: `MY_MACRO_THAT_SCARES_CPP_DEVELOPERS`.

```
#define ROUND(x) ...
#define PI_ROUNDED 5.0
```

Comments

dev-style-comments

Comments are vital to keeping our code readable. The following rules describe what you should comment and where. But remember: while comments are very important, the best code is self-documenting.

When writing your comments, write for your audience: the next contributor who will need to understand your code. Be generous — the next one may be you!

Nvim uses Doxygen comments.

Comment Style

Use the `//`-style syntax only.

```
// This is a comment spanning
// multiple lines
f();
```

File Comments

Start each file with a description of its contents.

Legal Notice

We have no such thing. These things are in LICENSE and only there.

File Contents

Every file should have a comment at the top describing its contents.

Generally a `.h` file will describe the variables and functions that are declared in the file with an overview of what they are for and how they are used. A `.c` file should contain more information about implementation details or discussions of tricky algorithms. If you feel the implementation details or a discussion of the algorithms would be useful for someone reading the `.h`, feel free to put it there instead, but mention in the `.c` that the documentation is in the `.h` file.

Do not duplicate comments in both the `.h` and the `.c`. Duplicated comments diverge.

```
/// A brief description of this file.
///
/// A longer description of this file.
/// Be very generous here.
```

Struct Comments

Every struct definition should have accompanying comments that describes what it is for and how it should be used.

```
/// Window info stored with a buffer.
///
/// Two types of info are kept for a buffer which are associated with a
/// specific window:
/// 1. Each window can have a different line number associated with a
/// buffer.
/// 2. The window-local options for a buffer work in a similar way.
/// The window-info is kept in a list at g_wininfo.  It is kept in
/// most-recently-used order.
struct win_info {
  /// Next entry or NULL for last entry.
  WinInfo *wi_next;
  /// Previous entry or NULL for first entry.
  WinInfo *wi_prev;
  /// Pointer to window that did the wi_fpos.
  Win *wi_win;
  ...
};
```

If the field comments are short, you can also put them next to the field. But be consistent within one struct, and follow the necessary doxygen style.

```
struct wininfo_S {
  WinInfo *wi_next;  ///< Next entry or NULL for last entry.
  WinInfo *wi_prev;  ///< Previous entry or NULL for first entry.
  Win *wi_win;       ///< Pointer to window that did the wi_fpos.
  ...
};
```

If you have already described a struct in detail in the comments at the top of your file feel free to simply state "See comment at top of file for a complete description", but be sure to have some sort of comment.

Document the synchronization assumptions the struct makes, if any. If an instance of the struct can be accessed by multiple threads, take extra care to document the rules and invariants surrounding multithreaded use.

Function Comments

Declaration comments describe use of the function; comments at the definition of a function describe operation.

Function Declarations

Every function declaration should have comments immediately preceding it that describe what the function does and how to use it. These comments should be descriptive ("Opens the file") rather than imperative ("Open the file"); the comment describes the function, it does not tell the function what to do. In general, these comments do not describe how the function performs its task. Instead, that should be left to comments in the function definition.

Types of things to mention in comments at the function declaration:

- If the function allocates memory that the caller must free.
- Whether any of the arguments can be a null pointer.
- If there are any performance implications of how a function is used.
- If the function is re-entrant. What are its synchronization assumptions?

```
/// Brief description of the function.
///
/// Detailed description.
/// May span multiple paragraphs.
///
/// @param arg1 Description of arg1
/// @param arg2 Description of arg2. May span
///        multiple lines.
///
/// @return Description of the return value.
Iterator *get_iterator(void *arg1, void *arg2);
```

Function Definitions

If there is anything tricky about how a function does its job, the function definition should have an explanatory comment. For example, in the definition comment you might describe any coding tricks you use, give an overview of the steps you go through, or explain why you chose to implement the function in the way you did rather than using a viable alternative. For instance, you might mention why it must acquire a lock for the first half of the function but why it is not needed for the second half.

Note you should not just repeat the comments given with the function declaration, in the `.h` file or wherever. It's okay to recapitulate briefly what the function does, but the focus of the comments should be on how it does it.

```
// Note that we don't use Doxygen comments here.
Iterator *get_iterator(void *arg1, void *arg2)
{
  ...
}
```

Variable Comments

In general the actual name of the variable should be descriptive enough to give a good idea of what the variable is used for. In certain cases, more comments are required.

Global Variables

All global variables should have a comment describing what they are and what they are used for. For example:

```
/// The total number of tests cases that we run
/// through in this regression test.
const int kNumTestCases = 6;
```

Implementation Comments

In your implementation you should have comments in tricky, non-obvious, interesting, or important parts of your code.

Line Comments

Also, lines that are non-obvious should get a comment at the end of the line. These end-of-line comments should be separated from the code by 2 spaces. Example:

```
// If we have enough memory, mmap the data portion too.
mmap_budget = max<int64>(0, mmap_budget - index_->length());
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock)) {
  return;  // Error already logged.
}
```

Note that there are both comments that describe what the code is doing, and comments that mention that an error has already been logged when the function returns.

If you have several comments on subsequent lines, it can often be more readable to line them up:

```
do_something();                     // Comment here so the comments line up.
do_something_else_that_is_longer();  // Comment here so there are two spaces between
                                    // the code and the comment.
{ // One space before comment when opening a new scope is allowed,
  // thus the comment lines up with the following comments and code.
  do_something_else();  // Two spaces before line comments normally.
}
```

NULL, true/false, 1, 2, 3...

When you pass in a null pointer, boolean, or literal integer values to functions, you should consider adding a comment about what they are, or make your code self-documenting by using constants. For example, compare:

```
bool success = calculate_something(interesting_value,
                                   10,
                                   false,
                                   NULL);  // What are these arguments??
```

versus:

```
bool success = calculate_something(interesting_value,
                                   10,     // Default base value.
                                   false,  // Not the first time we're calling this.
                                   NULL);  // No callback.
```

Or alternatively, constants or self-describing variables:

```
const int kDefaultBaseValue = 10;
const bool kFirstTimeCalling = false;
Callback *null_callback = NULL;
bool success = calculate_something(interesting_value,
                                   kDefaultBaseValue,
                                   kFirstTimeCalling,
                                   null_callback);
```

Don'ts

Note that you should never describe the code itself. Assume that the person reading the code knows C better than you do, even though he or she does not know what you are trying to do:

```
// Now go through the b array and make sure that if i occurs,
// the next element is i+1.
...        // Geez.  What a useless comment.
```

Punctuation, Spelling and Grammar

Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.

Comments should be as readable as narrative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the end of a line of code, can sometimes be less formal, but you should be consistent with your style.

Although it can be frustrating to have a code reviewer point out that you are using a comma when you should be using a semicolon, it is very important that source code maintain a high level of clarity and readability. Proper punctuation, spelling, and grammar help with that goal.

TODO Comments

Use `TODO` comments for code that is temporary, a short-term solution, or good-enough but not perfect.

TODOs should include the string TODO in all caps, followed by the name, email address, or other identifier of the person who can best provide context about the problem referenced by the TODO. The main purpose is to have a consistent TODO format that can be searched to find the person who can provide more details upon request. A TODO is not a commitment that the person referenced will fix the problem. Thus when you create a TODO, it is almost always your name that is given.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke): change this to use relations.
```

If your TODO is of the form "At a future date do something" make sure that you either include a very specific date ("Fix by November 2005") or a very specific event ("Remove this code when all clients can handle XML responses.").

Deprecation Comments

Mark deprecated interface points with `@deprecated` docstring token.

You can mark an interface as deprecated by writing a comment containing the word `@deprecated` in all caps. The comment goes either before the declaration of the interface or on the same line as the declaration.

After `@deprecated`, write your name, email, or other identifier in parentheses.

A deprecation comment must include simple, clear directions for people to fix their callsites. In C, you can implement a deprecated function as an inline function that calls the new interface point.

Marking an interface point DEPRECATED will not magically cause any callsites to change. If you want people to actually stop using the deprecated facility, you will have to fix the callsites yourself or recruit a crew to help you.

New code should not contain calls to deprecated interface points. Use the new interface point instead. If you cannot understand the directions, find the person who created the deprecation and ask them for help using the new interface point.

Formatting

dev-style-format

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

Non-ASCII Characters

Non-ASCII characters should be rare, and must use UTF-8 formatting.

You shouldn't hard-code user-facing text in source (OR SHOULD YOU?), even English, so use of non-ASCII characters should be rare. However, in certain cases it is appropriate to include such words in your code. For example, if your code parses data files from foreign sources, it may be appropriate to hard-code the non-ASCII string(s) used in those data files as delimiters. More commonly, unittest code (which does not need to be localized) might contain non-ASCII strings. In such cases, you should use UTF-8, since that is an encoding understood by most tools able to handle more than just ASCII.

Hex encoding is also OK, and encouraged where it enhances readability — for example, `"\uFEFF"`, is the Unicode zero-width no-break space character, which would be invisible if included in the source as straight UTF-8.

Braced Initializer Lists

Format a braced list exactly like you would format a function call in its place but with one space after the { and one space before the }

If the braced list follows a name (e.g. a type or variable name), format as if the {} were the parentheses of a function call with that name. If there is no name, assume a zero-length name.

```
struct my_struct m = {  // Here, you could also break before {.
    superlongvariablename1,
    superlongvariablename2,
    { short, interior, list },
    { interiorwrappinglist,
      interiorwrappinglist2 } };
```

Loops and Switch Statements

Annotate non-trivial fall-through between cases.

If not conditional on an enumerated value, switch statements should always have a `default` case (in the case of an enumerated value, the compiler will warn you if any values are not handled). If the default case should never execute, simply use `abort()`:

```
switch (var) {
  case 0:
    ...
    break;
  case 1:
    ...
    break;
  default:
    abort();
}
```

Switch statements that are conditional on an enumerated value should not have a `default` case if it is exhaustive. Explicit case labels are preferred over `default`, even if it leads to multiple case labels for the same code. For example, instead of:

```
case A:
  ...
case B:
  ...
case C:
  ...
default:
  ...
```

You should use:

```
case A:
  ...
case B:
  ...
case C:
  ...
case D:
case E:
case F:
  ...
```

Certain compilers do not recognize an exhaustive enum switch statement as exhaustive, which causes compiler warnings when there is a return statement in every case of a switch statement, but no catch-all return statement. To fix these spurious errors, you are advised to use `UNREACHABLE` after the switch statement to explicitly tell the compiler that the switch statement always returns and any code after it is unreachable. For example:

```
enum { A, B, C } var;
...
switch (var) {
  case A:
    return 1;
  case B:
    return 2;
  case C:
    return 3;
}
UNREACHABLE;
```

Return Values

Do not needlessly surround the `return` expression with parentheses.

Use parentheses in `return expr;` only where you would also use them in `x = expr;`.

```
return result;
return (some_long_condition && another_condition);

return (value);  // You wouldn't write var = (value);
return(result);  // return is not a function!
```

Horizontal Whitespace

Use of horizontal whitespace depends on location.

Variables

```
int long_variable = 0;  // Don't align assignments.
int i             = 1;

struct my_struct {  // Exception: struct arrays.
  const char *boy;
  const char *girl;
  int pos;
} my_variable[] = {
  { "Mia",       "Michael", 8  },
  { "Elizabeth", "Aiden",   10 },
  { "Emma",      "Mason",   2  },
};
```

Parting Words

The style guide is intended to make the code more readable. If you think you must violate its rules for the sake of clarity, do it! But please add a note to your pull request explaining your reasoning.

Writing tests for Nvim

Writing tests for Nvim

Nvim has a powerful yet simple test framework. It's approximately 7x better than whatever you use at work.

Each test starts a new Nvim process (10-30ms) which is discarded after the test finishes. You assert stuff using `t.eq()` and `screen:expect()` (which automatically waits as needed). That's pretty much it.

TODO: Expose the test framework as a public interface, for use in 3P plugins: https://github.com/neovim/neovim/issues/34592

Test framework

Tests are broadly divided into unit tests (`test/unit/`), functional tests (`test/functional/`) and old tests (`test/old/testdir/`).

- Unit testing is achieved by compiling the tests as a shared library which is loaded and called by [LuaJit FFI](https://luajit.org/ext_ffi.html).
- Functional tests are driven by RPC, so they do not require LuaJit (as opposed to Lua). They are essentially "integration" tests, they test the full system. But they are fast.

You can learn [Lua concepts 15 minutes](https://learnxinyminutes.com/docs/lua/), see also lua-guide. Use any existing test as a template to start writing new tests, or see dev-quickstart.

Tests are run by the `/cmake/RunTests.cmake` script using the repo-local Lua test harness in `/test/harness.lua`. For some failures, `./build/nvim.log` (or `$NVIM_LOG_FILE`) may provide insight.

Depending on the presence of binaries (e.g., `xclip`) some tests will be skipped.

Harness isolation dev-test-harness-isolation

The Lua harness runs all selected spec files in one `nvim -l` process, then restores a baseline before each file. For each suite iteration, the file boundary baseline is captured after `--helper` is loaded, so helper-provided modules and defaults persist across files. Helper files are preload-only: they may require modules, set defaults, and register suite-end cleanup, but they do not define tests or hooks.

The harness restores `_G`, `package.loaded`, `package.preload`, `arg`, and the process environment with the same shallow file boundary model that busted uses.

Test Layout

- `/test/benchmark` : benchmarks
- `/test/functional` : functional tests
- `/test/unit` : unit tests
- `/test/old/testdir` : old tests (from Vim)
- `/test/config` : contains `*.in` files which are transformed into `*.lua` files using `configure_file` CMake command: this is for accessing CMake variables in Lua tests.
- `/test/includes` : include-files for use by luajit `ffi.cdef` C definitions parser: normally used to make macros not accessible via this mechanism accessible the other way.
- `/test/*/preload.lua` : modules and defaults preloaded by the harness `--helper` option
- `/test/**/testutil.lua` : common utility functions in the context of the test runner
- `/test/**/testnvim.lua` : common utility functions in the context of the test session (RPC channel to the Nvim child process created by clear() for each test)
- `/test/*/**/*_spec.lua` : actual tests. Files that do not end with `_spec.lua` are libraries like `/test/**/testutil.lua`, except that they have some common topic.

Running tests dev-run-test

EXECUTING TESTS

To run all tests (except "old" tests):

```
make test
```

To run only _unit_ tests:

```
make unittest
```

To run only _functional_ tests:

```
make functionaltest
```

To run functional tests in parallel (used in CI):

```
cmake --build build --target functionaltest-parallel -j2
cmake --build build --target functionaltest-summary
```

LEGACY TESTS

To run all legacy Vim tests:

```
make oldtest
```

To run a _single_ legacy test file you can use either:

```
# Specify only the test file name, not the full path.
make oldtest TEST_FILE=test_syntax.vim
```

or:

```
make test/old/testdir/test_syntax.vim
```

DEBUGGING TESTS

- Each test gets a test id which looks like "T123". This also appears in the log file. Child processes spawned from a test appear in the logs with the _parent_ name followed by "/c". Example:

  ```
  DBG 2022-06-15T18:37:45.226 T57.58016.0   UI: flush
  DBG 2022-06-15T18:37:45.226 T57.58016.0   inbuf_poll:442: blocking... events_enabled=0 events_pending=0
  DBG 2022-06-15T18:37:45.227 T57.58016.0/c UI: stop
  INF 2022-06-15T18:37:45.227 T57.58016.0/c os_exit:595: Nvim exit: 0
  DBG 2022-06-15T18:37:45.229 T57.58016.0   read_cb:118: closing Stream (0x7fd5d700ea18): EOF (end of file)
  INF 2022-06-15T18:37:45.229 T57.58016.0   on_proc_exit:400: exited: pid=58017 status=0 stoptime=0
  ```

- You can set $GDB to [run functional tests under gdbserver](https://github.com/neovim/neovim/pull/1527):

  ```
  GDB=1 TEST_FILE=test/functional/api/buffer_spec.lua TEST_FILTER='nvim_buf_set_text works$' make functionaltest
  ```

  Read more about dev-filter-test.

Then, in another terminal:

```
gdb -ex 'target remote localhost:7777' build/bin/nvim
```

If $VALGRIND is also set it will pass `--vgdb=yes` to valgrind instead of starting gdbserver directly.

See `nvim_argv` in `test/functional/testnvim.lua`.

- Hanging tests can happen due to unexpected modal prompts (waiting for user input). The default screen width is 50 columns.
  - Commands that try to print lines longer than 50 columns in the cmdline, e.g. `:edit very...long...path`, will trigger the hit-enter prompt. Try using a shorter path, or `:silent edit`.
- If you can't figure out what is going on, try to visualize the screen. Put this at the beginning of your test:

  ```
  local Screen = require('test.functional.ui.screen')
  local screen = Screen.new()
  screen:attach()
  ```

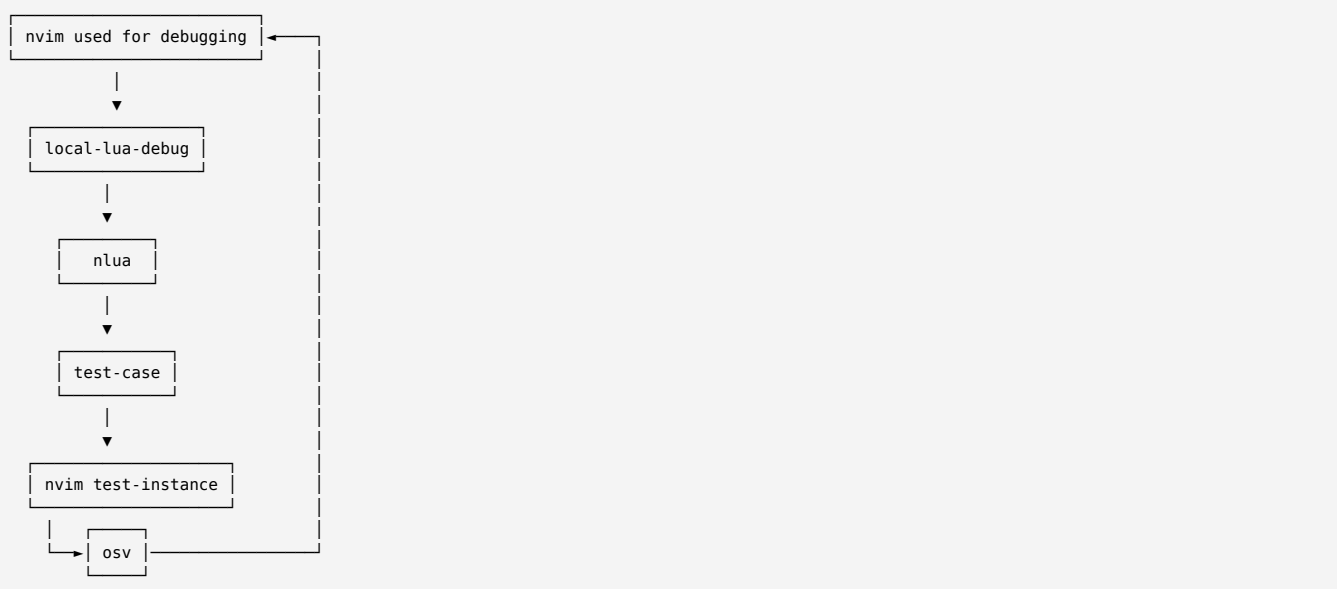
  Then put `screen:snapshot_util()` anywhere in your test. See the comments in `test/functional/ui/screen.lua` for more info.

DEBUGGING LUA TEST CODE

Debugging Lua test code is a bit involved. Get your shopping list ready, you'll need to install and configure:

1. [nvim-dap](https://github.com/mfussenegger/nvim-dap)
2. [local-lua-debugger-vscode](https://github.com/mfussenegger/nvim-dap/wiki/Debug-Adapter-installation#local-lua-debugger-vscode)
3. [nlua](https://github.com/mfussenegger/nlua)
4. [one-small-step-for-vimkind](https://github.com/jbyuki/one-small-step-for-vimkind) (called `osv`)
5. An `nlua` command in $PATH.

The setup roughly looks like this:



With these installed you can use a configuration like this:

```lua
local dap = require("dap")


local function free_port()
  local tcp = vim.loop.new_tcp()
  assert(tcp)
  tcp:bind('127.0.0.1', 0)
  local port = tcp:getsockname().port
  tcp:shutdown()
  tcp:close()
  return port
end


local name = "nvim-test-case" -- arbitrary name
local config = {
  name = name,

  -- value of type must match the key used in `dap.adapters["local-lua"] = ...` from step 2)
  type = "local-lua",

  request = "launch",
  cwd = "${workspaceFolder}",
  program = {
    command = "nlua",
  },
  args = {
    "-l",
    "test/runner.lua",
    "--helper=test/functional/preload.lua",
    "--lpath=build/?.lua",
    "--lpath=?.lua",

    -- path to file to debug, could be replaced with a hardcoded string
    function()
      return vim.api.nvim_buf_get_name(0)
    end,

    -- You can filter to specific test-case by adding:
    -- '--filter="' .. test_case_name .. '"',
  },
  env = {
    OSV_PORT = free_port
  }
}

-- Whenever the config is used it needs to launch a second debug session that attaches to `osv`
-- This makes it possible to step into `exec_lua` code blocks
setmetatable(config, {

  __call = function(c)
    ---@param session dap.Session
    dap.listeners.after.event_initialized["nvim_debug"] = function(session)
      if session.config.name ~= name then
        return
      end
      dap.listeners.after.event_initialized["nvim_debug"] = nil
      vim.defer_fn(function()
        dap.run({
          name = "attach-osv",
          type = "nlua", -- value must match the `dap.adapters` definition key for osv
          request = "attach",
          port = session.config.env.OSV_PORT,
        })
      end, 500)
    end

    return c
  end,
})
```

You can either add this configuration to your `dap.configurations.lua` list as described in `:help dap-configuration` or create it dynamically in a user-command or function and call it directly via `dap.run(config)`. The latter is useful if you use treesitter to find the test case around a cursor location with a query like the following and set the `--filter` property to it.

```
(function_call
  name: (identifier) @name (#any-of? @name "describe" "it")
  arguments: (arguments
    (string) @str
  )
)
```

Limitations:

- You need to add the following boilerplate to each spec file where you want to be able to stop at breakpoints within the test-case code:

```
if os.getenv("LOCAL_LUA_DEBUGGER_VSCODE") == "1" then
  require("lldebugger").start()
end
```

This is a local-lua-debugger limitation.

- You cannot step into code of files which get baked into the nvim binary (such as `_core/*.lua` and `inspect.lua`).

Filtering tests

dev-filter-test

FILTER BY NAME

Tests can be filtered by setting a pattern of test name to TEST_FILTER or TEST_FILTER_OUT.

```
it('foo api',function()
  ...
end)
it('bar api',function()
  ...
end)
```

To run only test with filter name:

```
TEST_FILTER='foo.*api' make functionaltest
```

To run all tests except ones matching a filter:

```
TEST_FILTER_OUT='foo.*api' make functionaltest
```

FILTER BY FILE

To run a _specific_ unit test:

```
TEST_FILE=test/unit/foo.lua make unittest
```

or

```
cmake -E env "TEST_FILE=test/unit/foo.lua" cmake --build build --target unittest
```

To run a _specific_ functional test:

```
TEST_FILE=test/functional/foo.lua make functionaltest
```

or

```
cmake -E env "TEST_FILE=test/functional/foo.lua" cmake --build build --target functionaltest
```

To _repeat_ a test:

```
TEST_ARGS="--repeat=100 --no-keep-going" TEST_FILE=test/functional/foo_spec.lua make functionaltest
```

or

```
cmake -E env "TEST_FILE=test/functional/foo_spec.lua" cmake -E env TEST_ARGS="--repeat=100 --no-keep-going" cmake --build build --target
functionaltest
```

FILTER BY TAG

Tests can be "tagged" by adding # before a token in the test description.

```
it('#foo bar baz', function()
  ...
end)
it('#foo another test', function()
  ...
end)
```

To run only the tagged tests:

```
TEST_TAG=foo make functionaltest
```

NOTE:

- TEST_FILE is not a pattern string like TEST_TAG or TEST_FILTER. The given value to TEST_FILE must be a path to an existing file.
- Both TEST_TAG and TEST_FILTER filter tests by the string descriptions found in `it()` and `describe()`.

Writing tests

dev-write-test

GUIDELINES

- CI runs tests in parallel via the `functionaltest-parallel` cmake target.
- The Nvim test instance CWD is isolated to per "test group" (see WORKING_DIRECTORY in `cmake/RunTests.cmake`), so tests can write files to CWD without interfering with other tests running in parallel.

- Luajit needs to know about type and constant declarations used in function prototypes. The [testutil.lua](https://github.com/neovim/neovim/blob/master/test/unit/testutil.lua) file automatically parses `types.h`, so types used in the tested functions could be moved to it to avoid having to rewrite the declarations in the test files.
  - `#define` constants must be rewritten `const` or `enum` so they can be "visible" to the tests.
- Use `pending()` or `t.skip()` to skip tests.
  - Example: https://github.com/neovim/neovim/commit/5c1dc0fbe7388528875aff9d7b5055ad718014de#diff-bf80b24c724b0004e8418102f68b0679R18
  - **Note:** If a test is skipped because of a non-actionable reason, we don't want it to appear in the "pending" list. Include "N/A" in the skip description, then it won't be added to the "pending" list. For example, if a test is skipped because it's running on a non-LuaJit system, including it in the "pending" list is just noise. Thus, its pending reason should say "N/A":

    ```
    pending('N/A: missing LuaJIT FFI')
    ```

  - Do not silently skip the test with `if-else`. If a functional test depends on some external factor (e.g. the existence of `md5sum` on `$PATH`), _and_ you can't mock or fake the dependency, then skip the test via `pending()` if the external factor is missing. This ensures that the _total_ test-count (success + fail + error + pending) is the same in all environments.
    - **Note:** `pending()` is ignored if it is missing an argument, unless it is [contained in an `it()` block](https://github.com/neovim/neovim/blob/d21690a66e7eb5ebef18046c7a79ef898966d786/test/functional/ex_cmds/grep_spec.lua#L11) Provide empty function argument if the `pending()` call is outside `it()` Example: https://github.com/neovim/neovim/commit/5c1dc0fbe7388528875aff9d7b5055ad718014de#diff-bf80b24c724b0004e8418102f68b0679R18

WHERE TESTS GO

Tests in `/test/unit` and `/test/functional` are divided into groups by the semantic component they are testing.

- Unit tests (`test/unit/`) should match 1-to-1 with the structure of `src/nvim/`, because they are testing functions directly. E.g. unit-tests for `src/nvim/undo.c` should live in `test/unit/undo_spec.lua`.
- Functional tests (`test/functional/`) are higher-level (plugins, UI, user input) than unit tests; they are organized by concept.
  - Try to find an existing `test/functional/*/*_spec.lua` group that makes sense, before creating a new one.

Fixing tests

dev-fix-test

FIXING HARNESS WARNINGS

> Nvim session T123 took 2000 milliseconds to exit > This indicates a likely problem with the test even if it passed!

This may indicate a leak, because Nvim waits on uv handles before exiting. Example: https://github.com/neovim/neovim/pull/35768

FIXING LINT FAILURES

`make emmylua-check` runs EmmyLua on the runtime Lua code, using `.emmyrc.json`. The checker is downloaded automatically. Enable more checks incrementally by removing codes from `diagnostics.disable` and fixing the resulting diagnostics.

If an EmmyLua warning must be ignored, specify the warning code. Example:

```
---@diagnostic disable-next-line: unused
```

https://github.com/EmmyLuaLs/emmylua-analyzer-rust

Ignore the smallest applicable scope (e.g. inside a function, not at the top of the file).

Configuration

dev-test-config

(TODO: clean this up, too many variables and some of them are not used anymore.)

Test behaviour is affected by environment variables. Currently supported (Functional, Unit, Benchmarks) (when Defined; when set to _1_; when defined, treated as Integer; when defined, treated as String; when defined, treated as Number; !must be defined to function properly):

- `TEST_ARGS` (F) (U): arguments forwarded to the local test harness.
- `CC` (U) (S): specifies which C compiler to use to preprocess files. Currently only compilers with gcc-compatible arguments are supported.
- `GDB` (F) (D): makes nvim instances to be run under `gdbserver`. It will be accessible on `localhost:7777`: use `gdb build/bin/nvim`, type `target remote :7777` inside.
- `GDBSERVER_PORT` (F) (I): overrides port used for `GDB`.
- `LOG_DIR` (FU) (S!): specifies where to seek for valgrind and ASAN log files.
- `VALGRIND` (F) (D): makes nvim instances to be run under `valgrind`. Log files are named `valgrind-%p.log` in this case. Note that non-empty valgrind log may fail tests. Valgrind arguments may be seen in `/test/functional/testnvim.lua`. May be used in conjunction with `GDB`.
- `VALGRIND_LOG` (F) (S): overrides valgrind log file name used for `VALGRIND`.
- `TEST_COLORS` (F) (U) (D): enable pretty colors in test runner. Set to true by default.
- `TEST_SKIP_FRAGILE` (F) (D): makes test suite skip some fragile tests.
- `TEST_TIMEOUT` (FU) (I): specifies maximum time, in seconds, before the test suite run is killed
- `NVIM_TEST` (FU) (D): lets Nvim process detect that it is running in a test.
- `NVIM_LUA_NOTRACK` (F) (D): disable reference counting of Lua objects
- `NVIM_PRG` (F) (S): path to Nvim executable (default: `build/bin/nvim`).
- `NVIM_TEST_MAIN_CDEFS` (U) (1): makes `ffi.cdef` run in main process. This raises a possibility of bugs due to conflicts in header definitions, despite the counters, but greatly speeds up unit tests by not requiring `ffi.cdef` to do parsing of big strings with C definitions.
- `NVIM_TEST_PRINT_I` (U) (1): makes `cimport` print preprocessed, but not yet filtered through `formatc` headers. Used to debug `formatc`. Printing is done with the line numbers.

- NVIM_TEST_PRINT_CDEF (U) (1): makes `cimport` print final lines which will be then passed to `ffi.cdef`. Used to debug errors `ffi.cdef` happens to throw sometimes.
- NVIM_TEST_PRINT_SYSCALLS (U) (1): makes it print to stderr when syscall wrappers are called and what they returned. Used to debug code which makes unit tests be executed in separate processes.
- NVIM_TEST_RUN_FAILING_TESTS (U) (1): makes `itp` run tests which are known to fail (marked by setting third argument to `true`).
- NVIM_TEST_CORE_* (FU) (S): a set of environment variables which specify where to search for core files. Are supposed to be defined all at once.
- NVIM_TEST_CORE_GLOB_DIRECTORY (FU) (S): directory where core files are located. May be `.`. This directory is then recursively searched for core files. **Note:** this variable must be defined for any of the following to have any effect.
- NVIM_TEST_CORE_GLOB_RE (FU) (S): regular expression which must be matched by core files. E.g. `/core[^/]*$`. May be absent, in which case any file is considered to be matched.
- NVIM_TEST_CORE_EXC_RE (FU) (S): regular expression which excludes certain directories from searching for core files inside. E.g. use `^/%.deps$` to not search inside `/.deps`. If absent, nothing is excluded.
- NVIM_TEST_CORE_DB_CMD (FU) (S): command to get backtrace out of the debugger. It may use environment variables _NVIM_TEST_APP (path to application which is being debugged: normally either nvim or luajit) and _NVIM_TEST_CORE (core file to get backtrace from). Default:

```
gdb -n -batch -ex "thread apply all bt full" "$_NVIM_TEST_APP" -c "$_NVIM_TEST_CORE"
```

- NVIM_TEST_CORE_RANDOM_SKIP (FU) (D): makes `check_cores` not check cores after approximately 90% of the tests. Should be used when finding cores is too hard for some reason. Normally (on OS X or when NVIM_TEST_CORE_GLOB_DIRECTORY is defined and this variable is not) cores are checked for after each test.
- NVIM_TEST_INTEG (F) (D): enables integration tests that makes real network calls. By default these tests are skipped. When set to 1, tests requiring external HTTP requests (e.g `vim.net.request()`) will be run.
- NVIM_TEST_RUN_TESTTEST (U) (1): allows running `test/unit/testtest_spec.lua` used to check how testing infrastructure works.
- NVIM_TEST_TRACE_LEVEL (U) (N): specifies unit tests tracing level:
  - `0` disables tracing (the fastest, but you get no data if tests crash and no core dump was generated),
  - 1 leaves only C function calls and returns in the trace (faster than recording everything),
  - 2 records all function calls, returns and executed Lua source lines.
- NVIM_TEST_TRACE_ON_ERROR (U) (1): makes unit tests yield trace on error in addition to regular error message.
- NVIM_TEST_MAXTRACE (U) (N): specifies maximum number of trace lines to keep. Default is 1024.
- OSV_PORT: (F): launches `osv` listening on the given port within nvim test instances.

Nvim colorscheme guidelines

Style guidelines for developing Nvim's default colorscheme.

License: CC-By 3.0 https://creativecommons.org/licenses/by/3.0/

Design

- Be "Neovim branded", i.e. have mostly "green-blue" feel plus one or two colors reserved for very occasional user attention.
- Be oriented for 'termguicolors' (true colors) while being extra minimal for 'notermguicolors' (16 colors) as fallback.
- Be accessible, i.e. have high enough contrast ratio (as defined in https://www.w3.org/TR/2008/REC-WCAG20-20081211/#contrast-ratiodef). This means to have value at least 7 for hl-Normal and 4.5 for some common cases (hl-Visual, `Comment` with set 'cursorline', colored syntax, `Diff*`, hl-Search).
- Be suitable for dark and light backgrounds via exchange of dark and light palettes.
- Be usable, i.e. provide enough visual feedback for common objects.

Palettes

- There are two separate palettes: dark and light. They all contain the same set of colors exported as `NvimDark*` and `NvimLight*` colors respectively.
- The dark palette is used for background in the dark color scheme and for foreground in the light color scheme; and vice versa. This introduces recognizable visual system without too standing out.
- Actual computation of palettes should be done in a perceptually uniform color space. Oklch is a good choice.
- Each palette has the following colors (descriptions are for dark background; reverse for light one):
  - Four shades of colored "cold" greys for general UI.
    - Dark ones (from darkest to lightest) are reserved as background for hl-NormalFloat (considered as "black"), hl-Normal (background), hl-CursorLine, hl-Visual.
    - Light ones (also from darkest to lightest) are reserved for `Comment`, hl-StatusLine/|hl-TabLine|, hl-Normal (foreground), and color considered as "white".
- Six colors to provide enough terminal colors: red, yellow, green, cyan, blue, magenta. They should have (reasonably) similar lightness and chroma to make them visually coherent. Lightness should be as equal to the palette's basic grey (which is used for hl-Normal) as possible. They should have (reasonably) different hues to make them visually separable.
- For 16 colors:
  - Greys are not used and are replaced with the foreground and background colors of the terminal emulator.
  - Non-grey colors fall back to terminal colors as ordered in ANSI codes (https://en.wikipedia.org/wiki/ANSI_escape_code#3-bit_and_4-bit), that is red (1, 9), green (2, 10), yellow (3, 11), blue (4, 12), magenta (5, 13), cyan (6, 14). To increase contrast, colors 1-6 are used for light background and 9-14 for dark background.

Highlight groups

Use:

- Grey shades for general UI according to their design.
- Bold text for keywords (`Statement` highlight group). This is an important choice to increase accessibility for people with color deficiencies, as it doesn't rely on actual color.
- Green for strings, hl-DiffAdd (as background), hl-DiagnosticOk, and some minor text UI elements.
- Cyan as main syntax color, i.e. for function usage (`Function` highlight group), hl-DiffText, hl-DiagnosticInfo, and some minor text UI elements.
- Red to generally mean high user attention, i.e. errors; in particular for hl-ErrorMsg, hl-DiffDelete, hl-DiagnosticError.
- Yellow very sparingly to mean mild user attention, i.e. warnings. That is, hl-DiagnosticWarn and hl-WarningMsg.
- Blue very sparingly as hl-DiagnosticHint and some additional important syntax group (like `Identifier`).
- Magenta very carefully (if at all).

In case of 16 colors:

- Rely on the assumption "Background color can be used as background; other colors can be used as foreground". This means that in any foreground/background combination there should be background and one non-background color.
- Use 0 (black) or 15 (bright white) as foreground for non-grey background, depending on whether normal background is light or dark.

Tools and techniques for developing Nvim

This is for developing or debugging Nvim itself.

Quickstart guide to developing Nvim dev-quickstart

You can start hacking on Nvim in less than 5 minutes:

1. Ensure you have the build prerequisites from `BUILD.md`.
2. Clone the source repo and "cd" into it:

```
git clone https://github.com/neovim/neovim
cd neovim
# (Optional) Build and run Nvim:
make
VIMRUNTIME=./runtime ./build/bin/nvim --luamod-dev
```

3. Run a single test. We will start with "example_spec.lua", which is a real test that shows how tests are written:

```
make functionaltest TEST_FILE=test/functional/example_spec.lua
```

4. Notice the `before_each` block in the test file. Because it calls `clear()`, each `it()` test will start a new Nvim instance.
5. Tests will do stuff in the Nvim instance and make assertions using `eq()`. Tests that want to check the UI can also use `screen:expect()`.
6. Now make a code change in Nvim itself, then you can see the effects. The example test does `feed('iline1…')`, so let's make a change to the insert-mode code, which lives in `src/nvim/insert.c`. In the `insert_handle_key` function, just after the `normalchar` label, add this code:

```
s->c = 'x';
```

7. Then run the "example_spec.lua" test again, and it should fail with something like this:

```
test/functional/example_spec.lua:31: Row 1 did not match.
Expected:
  |*line1              |
  |*line^2              |
  |{0:~                  }|
  |{0:~                  }|
  |                     |
Actual:
  |*xine1              |
  |*xine^2              |
  |{0:~                  }|
  |{0:~                  }|
  |                     |
```

You now understand how to modify the codebase, write tests, and run tests. See dev-arch for details about the internal architecture.

Logs dev-tools-logs

Low-level log messages sink to `$NVIM_LOG_FILE`.

UI events are logged at DEBUG level.

```
rm -rf build/
make CMAKE_BUILD_TYPE=Debug
```

Use `LOG_CALLSTACK()` (Linux only) to log the current stacktrace. To log to an alternate file (e.g. stderr) use `LOG_CALLSTACK_TO_FILE(FILE*)`. Requires `-no-pie` ([ref](https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=860394#15)):

```
rm -rf build/
make CMAKE_BUILD_TYPE=Debug CMAKE_EXTRA_FLAGS="-DCMAKE_C_FLAGS=-no-pie"
```

Many log messages have a shared prefix, such as "UI" or "RPC". Use the shell to filter the log, e.g. at DEBUG level you might want to exclude UI messages:

```
tail -F ~/.local/state/nvim/nvim.log | cat -v | stdbuf -o0 grep -v UI | stdbuf -o0 tee -a log
```

Reproducible build

To make a reproducible build of Nvim, set cmake variable LUA_GEN_PRG to a LuaJIT binary built with `LUAJIT_SECURITY_PRN=0`. See commit cb757f2663e6950e655c6306d713338dfa66b18d.

Debug TUI dev-tools-tui

TUI INSPECT

Use the Ghostty https://ghostty.org/ inspector tool to observe and query the output and events from any terminal application such as Nvim.

From the Ghostty inspector you can click the "Terminal IO" tab to get a trace.

TERMINFO LOGGING

At 'verbose' level 3, Nvim logs its internal terminfo state, so you can see exactly what terminfo values it is using on the current system.

```
nvim -V3log
```

TUI DEBUGGING WITH GDB LLDB

Launching the Nvim TUI involves two processes, one for main editor state and one for rendering the TUI. Both of these processes use the nvim binary, so somewhat confusingly setting a breakpoint in either will generally succeed but may not be hit depending on which process the breakpoints were set in.

To debug the main process, you can debug the nvim binary with the `--headless` flag which does not launch the TUI and will allow you to set breakpoints in code not related to TUI rendering like so:

```
lldb -- ./build/bin/nvim --headless --listen ~/.cache/nvim/debug-server.pipe
```

While in lldb, enter `run`. You can then attach to the headless process in a new terminal window to interact with the editor like so:

```
./build/bin/nvim --remote-ui --server ~/.cache/nvim/debug-server.pipe
```

Conversely for debugging TUI rendering, you can start a headless process and debug the remote-ui process multiple times without losing editor state.

For details on using nvim-dap and automatically debugging the child (main) process, see [here](https://zignar.net/2023/02/17/debugging-neovim-with-neovim-and-nvim-dap/)

TUI REDRAW

For debugging Nvim TUI redraw behavior it is sometimes useful to slow down its redraws. Set the 'writedelay' and 'redrawdebug' options to see where and when the UI is painted.

```
:set writedelay=50 rdb=compositor
```

Note: Nvim uses an internal screenbuffer to only send minimal updates even if a large region is repainted internally. To also highlight excess internal redraws, use

```
:set writedelay=50 rdb=compositor,nodelta
```

TUI TRACE

From the Ghostty inspector you can click the "Terminal IO" tab to get a trace.

Alternatively, the ancient `script` command is the "state of the art". The libvterm `vterm-dump` utility formats the result for human-readability.

Record a Nvim terminal session and format it with `vterm-dump`:

```
script foo
./build/bin/nvim -u NONE
# Exit the script session with CTRL-d

# Use `vterm-dump` utility to format the result.
./.deps/usr/bin/vterm-dump foo > bar
```

Then you can compare `bar` with another session, to debug TUI behavior.

TERMINAL REFERENCE

- `man terminfo`
- https://github.com/vim/vim/blob/0124320c97b0fbbb44613f42fc1c34fee6181fc8/src/libvterm/doc/seqs.txt
- https://invisible-island.net/xterm/ctlseqs/ctlseqs.html

Debug Performance

dev-tools-perf

PROFILING (EASY)

For debugging performance bottlenecks in any code, there is a simple (and very effective) approach:

1. Run the slow code in a loop.
2. Break execution ~5 times and save the stacktrace.
3. The cause of the bottleneck will (almost always) appear in most of the stacktraces.

PROFILING (FANCY)

For more advanced profiling, consider `perf` + `flamegraph`.

USDT PROFILING (POWERFUL)

Or you can use USDT probes via `NVIM_PROBE` ([#12036](https://github.com/neovim/neovim/pull/12036)).

> USDT is basically a way to define stable probe points in userland binaries. > The benefit of bcc is the ability to define logic to go along with the probe > points.

Tools:
- bpftrace provides an awk-like language to the kernel bytecode, BPF.
- BCC provides a subset of C. Provides more complex logic than bpftrace, but takes a bit more effort.

Example using bpftrace to track slow vim functions, and print out any files that were opened during the trace. At the end, it prints a histogram of function timing:

```
#!/usr/bin/env bpftrace

BEGIN {
  @depth = -1;
}

tracepoint:sched:sched_process_fork /@pidmap[args->parent_pid]/ {
  @pidmap[args->child_pid] = 1;
}

tracepoint:sched:sched_process_exit /@pidmap[args->pid]/ {
  delete(@pidmap[args->pid]);
}

usdt:build/bin/nvim:neovim:eval__call_func__entry {
    @pidmap[pid] = 1;
    @depth++;
    @funcentry[@depth] = nsecs;
}

usdt:build/bin/nvim:neovim:eval__call_func__return {
    $func = str(arg0);
    $msecs = (nsecs - @funcentry[@depth]) / 1000000;

    @time_histo = hist($msecs);

    if ($msecs >= 1000) {
      printf("%u ms for %s\n", $msecs, $func);
      print(@files);
    }

    clear(@files);
    delete(@funcentry[@depth]);
    @depth--;
}

tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat {
  if (@pidmap[pid] == 1 && @depth >= 0) {
    @files[str(args->filename)] = count();
  }
}

END {
  clear(@depth);
}

$ sudo bpftrace funcslower.bt
1527 ms for Slower
@files[/usr/lib/libstdc++.so.6]: 2
@files[/etc/fish/config.fish]: 2
<snip>

^C
@time_histo:
[0]                71430 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[1]                  346 |                                                     |
[2, 4)               208 |                                                     |
[4, 8)                91 |                                                     |
[8, 16)               22 |                                                     |
[16, 32)              85 |                                                     |
[32, 64)               7 |                                                     |
[64, 128)              0 |                                                     |
[128, 256)             0 |                                                     |
[256, 512)             6 |                                                     |
[512, 1K)              1 |                                                     |
[1K, 2K)               5 |                                                     |
```

Backtraces

dev-tools-backtrace

LINUX

Core dumps are disabled by default on Ubuntu, CentOS and others. To enable core dumps:

```
ulimit -c unlimited
```

On systemd-based systems getting a backtrace is as easy as:

```
coredumpctl -1 gdb
```

`coredumpctl` is an optional tool, so you may need to install it:

```
sudo apt install systemd-coredump
```

The full backtrace is most useful; please send us the `backtrace.txt` file when reporting a bug related to a crash:

```
2>&1 coredumpctl -1 gdb | tee -a backtrace.txt
(gdb) thread apply all bt full
```

On systems without `coredumpctl`, you may find a `core` dump file appearing in the current directory or in other locations. On Linux systems where `apport` is installed (such as Ubuntu), the directory where core dump files are saved can be `/var/lib/apport/coredump` or elsewhere, depending on the system configuration (see `/proc/sys/kernel/core_pattern`). See also: https://stackoverflow.com/a/18368068

To get a backtrace from the `./core` dump file:

```
gdb build/bin/nvim ./core 2>&1 | tee backtrace.txt
(gdb) thread apply all bt full
```

MACOS

If `nvim` crashes, you can see the backtrace in `Console.app` (under "Crash Reports" or "User Diagnostic Reports" for older macOS versions).

```
open -a Console
```

You may also want to enable core dumps on macOS. To do this, first make sure the `/cores/` directory exists and is writable:

```
sudo mkdir /cores
sudo chown root:admin /cores
sudo chmod 1775 /cores
```

Then set the core size limit to `unlimited`:

```
ulimit -c unlimited
```

Note that this is done per shell process. If you want to make this the default for all shells, add the above line to your shell's init file (e.g. `~/.bashrc` or similar).

You can then open the core file in `lldb`:

```
lldb -c /cores/core.12345
```

Apple's documentation archive has some other useful information https://developer.apple.com/library/archive/technotes/tn2124/_index.html#//apple_ref/doc/uid/DTS10003391-CH1-SECCOREDUMPS, but note that some of the things on this page are out of date (such as enabling core dumps with `/etc/launchd.conf`).

WINDOWS

If the Windows version of Nvim crashes in a reproducible manner, you can take some steps to provide a useful bug report.

You must obtain the debugger symbols (PDB) file for the Nvim executable: nvim.pdb. The PDB should be available from the same place that you obtained the executable (TODO: not currently provided by Nvim CI releases). Be sure to use the PDB that matches the EXE (same build).

If you built the executable yourself with the Microsoft Visual C++ compiler, then the PDB was built with the EXE.

If you have Visual Studio, use that instead of the VC Toolkit and WinDbg.

For other compilers, always use the corresponding debugger: gdb or lldb.

Debugging Nvim crashes with Visual Studio 2005

First launch nvim.exe and then launch Visual Studio. (If you don't have Visual Studio, get it from https://visualstudio.microsoft.com/downloads/).

On the Tools menu, click Attach to Process. Choose the Nvim process.

In Nvim, reproduce the crash. A dialog will appear in Visual Studio, telling you about the unhandled exception in the Nvim process. Click Break to break into the process.

Visual Studio will pop up another dialog, telling you that no symbols are loaded and that the source code cannot be displayed. Click OK.

Several windows will open. Right-click in the Call Stack window. Choose Load Symbols. The Find Symbols dialog will open, looking for (g)vim.pdb. Navigate to the directory where you have the PDB file and click Open.

At this point, you should have a full call stack with vim function names and line numbers. Double-click one of the lines and the Find Source dialog will navigate to the directory where the Nvim source is (if you have it.)

If you don't know how to debug this any further, follow the instructions at ":help bug-report". Paste the call stack into the bug report.

From Visual Studio you can also try saving a minidump via the Debug menu and send it with the bug report. A minidump is a small file (<100KB), which contains information about the state of your process.

Gdb

dev-tools-gdb

USING GDB TO STEP THROUGH FUNCTIONAL TESTS

Use TEST_TAG to run tests matching `#foo` tags (e.g. `it("test #foo ...", ...)`):

```
GDB=1 TEST_TAG=foo make functionaltest
```

Then, in another terminal:

```
gdb build/bin/nvim
(gdb) target remote localhost:7777
```

-- See `nvim_argv` in https://github.com/neovim/neovim/blob/master/test/functional/testnvim.lua.

USING LLDB TO STEP THROUGH UNIT TESTS

```
lldb build/bin/nvim -- -l test/runner.lua --lpath=./build/?.lua test/unit/
```

USING GDB

To attach to a running `nvim` process with a pid of 1234 (Tip: the pid of a running Nvim instance can be obtained by calling getpid()), for instance:

```
gdb -tui -p 1234 build/bin/nvim
```

The gdb interactive prompt will appear. At any time you can:

- `break foo` to set a breakpoint on the `foo()` function
- n to step over the next statement
  - <Enter> to repeat the last command
- s to step into the next statement
- c to continue
- `finish` to step out of the current function
- `p zub` to print the value of zub
- bt to see a backtrace (callstack) from the current location
- `CTRL-x CTRL-a` or `tui enable` to show a TUI view of the source file in the current debugging context. This can be extremely useful as it avoids the need for a gdb "frontend".
- <up> and <down> to scroll the source file view

GDB REVERSE DEBUGGING

- `set record full insn-number-max unlimited`
- `continue` for a bit (at least until `main()` is executed
- `record`
- provoke the bug, then use `revert-next`, `reverse-step`, etc. to rewind the debugger

USING GDBSERVER

You may want to connect multiple gdb clients to the same running `nvim` process, or you may want to connect to a remote `nvim` process with a local gdb. Using `gdbserver`, you can attach to a single process and control it from multiple gdb clients.

Open a terminal and start `gdbserver` attached to `nvim` like this:

```
gdbserver :6666 build/bin/nvim 2> gdbserver.log
```

`gdbserver` is now listening on port 6666. You then need to attach to this debugging session in another terminal:

```
gdb build/bin/nvim
```

Once you've entered gdb, you need to attach to the remote session:

```
(gdb) target remote localhost:6666
```

In case gdbserver puts the TUI as a background process, the TUI can become unable to read input from pty (and receives SIGTTIN signal) and/or output data (SIGTTOU signal). To force the TUI as the foreground process, you can add

```
signal (SIGTTOU, SIG_IGN);
if (!tcsetpgrp(data->input.in_fd, getpid())) {
    perror("tcsetpgrp failed");
}
```

to `tui.c:terminfo_start`.

USING GDBSERVER IN TMUX

Consider using a custom makefile https://github.com/neovim/neovim/blob/master/BUILD.md#custom-makefile to quickly start debugging sessions using the `gdbserver` method mentioned above. This example `local.mk` will create the debugging session when you type `make debug`.

```
.PHONY: dbg-start dbg-attach debug build

build:
    @$(MAKE) nvim

dbg-start: build
    @tmux new-window -n 'dbg-neovim' 'gdbserver :6666 ./build/bin/nvim -D'

dbg-attach:
    @tmux new-window -n 'dbg-cgdb' 'cgdb -x gdb_start.sh ./build/bin/nvim'

debug: dbg-start dbg-attach
```

Here `gdb_start.sh` includes gdb commands to be called when the debugger starts. It needs to attach to the server started by the `dbg-start` rule. For example:

```
(gdb) target remote localhost:6666
(gdb) br main
```

Debugging crashes or memory leaks

dev-tools-asan

BUILD WITH ASAN

Building Nvim with Clang sanitizers (Address Sanitizer: ASan, Undefined Behavior Sanitizer: UBSan, Memory Sanitizer: MSan, Thread Sanitizer: TSan) is a good way to catch undefined behavior, leaks and other errors as soon as they happen. It's significantly faster than Valgrind.

Requires clang 3.4 or later, and `llvm-symbolizer` must be in `$PATH`:

```
clang --version
```

Build Nvim with sanitizer instrumentation (choose one):

```
CC=clang make CMAKE_EXTRA_FLAGS="-DENABLE_ASAN_UBSAN=ON"
CC=clang make CMAKE_EXTRA_FLAGS="-DENABLE_MSAN=ON"
CC=clang make CMAKE_EXTRA_FLAGS="-DENABLE_TSAN=ON"
```

Create a directory to store logs:

```
mkdir -p "$HOME/logs"
```

Configure the sanitizer(s) via these environment variables:

```
# Change to detect_leaks=1 to detect memory leaks (slower, noisier).
export ASAN_OPTIONS="detect_leaks=0:log_path=$HOME/logs/asan"
# Show backtraces in the logs.
export MSAN_OPTIONS="log_path=${HOME}/logs/msan"
export TSAN_OPTIONS="log_path=${HOME}/logs/tsan"
```

Logs will be written to `${HOME}/logs/*san.PID` then.

For more information: https://github.com/google/sanitizers/wiki/SanitizerCommonFlags

If you run nvim in the builtin TUI, it can be asked to load the crash log after a crash as a quickfix list in a restarted nvim instance. Put this in init.lua (adjust quickfix commands to taste):

```
if vim.env.ASAN_OPTIONS ~= nil then
  local log_name = vim.env.HOME.."/logs/asan."..vim.uv.getpid()
  local args = {"--embed", "-n", "+set efm=%+A%*[^/]%f:%l:%c", "+silent cfile "..log_name, "+silent cfirst", "+silent copen"}
  vim.api.nvim__set_restart_on_crash("nvim", args)
end
```

Merging patches from Vim

Nvim was forked from Vim 7.4.160; it is kept up-to-date with relevant Vim patches in order to avoid duplicate work. Run `vim-patch.sh` to see the status of Vim patches:

```
./scripts/vim-patch.sh -l
```

Everyone is welcome to create a dev-vimpatch-pr for relevant Vim patches, but some types of patches are dev-vimpatch-na. See dev-vimpatch-quickstart to get started immediately.

QUICKSTART — dev-vimpatch-quickstart

1. Pull the Nvim source:

```
git clone https://github.com/neovim/neovim.git
```

2. Run `./scripts/vim-patch.sh -l` to see the list of missing Vim patches.
3. Choose a patch from the list (usually the oldest one), e.g. `8.0.0123`.
   - Check for open vim-patch PRs https://github.com/neovim/neovim/pulls?q=is%3Apr+is%3Aopen+label%3Avim-patch.
4. Run `./scripts/vim-patch.sh -p 8.0.0123`
5. Follow the instructions given by the script.

NOTES

- It's strongly recommended to work on the oldest missing patch, because later patches might depend on the changes.
- Use `git log --grep` or `git log -G` to search the Nvim/Vim source history (even deleted code). E.g. to find `reset_option_was_set`:

```
git log -p -G reset_option_was_set
```

- Pass `git log` options like `--grep` and `-G` to `./scripts/vim-patch.sh -L` to filter unmerged Vim patches E.g. to find `+quickfix` patches:

```
./scripts/vim-patch.sh -L --grep quickfix -- src
```

PULL REQUESTS — dev-vimpatch-pr

Note: vim-patch.sh automates these steps for you. Use it!

- Install gh (https://cli.github.com/) if you want to use `vim-patch.sh` to create PRs automatically
- The pull request title should include `vim-patch:8.x.xxxx` (no whitespace)
- The commit message https://github.com/neovim/neovim/commit/4ccf1125ff569eccfc34abc4ad794044c5ab7455 should include:
  - A token indicating the Vim patch number, formatted as follows: `vim-patch:8.0.0123` (no whitespace)
  - A URL pointing to the Vim commit: https://github.com/vim/vim/commit/c8020ee825b9d9196b1329c0e097424576fc9b3a
  - The original Vim commit message, including author

Reviewers: hint for reviewing `runtime/` patches https://github.com/neovim/neovim/pull/1744#issuecomment-68202876

NA (NOT APPLICABLE) PATCHES — dev-vimpatch-na

Most Vim patches are not applicable to Nvim. If you find NA patches, visit an the latest "version.c: update" pull request https://github.com/neovim/neovim/pulls?q=is%3Apr+author%3Aapp%2Fgithub-actions+version.c+is%3Aopen and mention the NA patches in a comment (please edit/update one comment, rather than adding a new comment for each patch).

Or if there is no open `version.c: update` pull request, include NA patches in a commit message in this format:

```
vim-patch:<version-or-commit>
vim-patch:<version-or-commit>
...
```

where `<version-or-commit>` is a valid Vim version (like `8.0.0123`) or commit-id (SHA). Each patch is on a separate line.

Example: https://github.com/neovim/neovim/commit/00f60c2ce78fc1280e93d5a36bc7b2267d5f4ac6

HOW TO DECIDE IF A VIM-PATCH IS WANTED

Legacy Vim is becoming much less relevant to Nvim: we expect to decline the majority of their patches. Thus this is an "allow-list" instead of a "deny-list". Generally, only modules explicitly noted in the "OWNED BY VIM" list below, are in-scope; everything else should be marked "N/A".

OWNED BY VIM

- quickfix
- regex
- runtime files

Everything else is owned by Nvim. Most of Vim's "bug fixes" for e.g. buffer/window behavior, or the events/autocmds subysytem, or "getchar" (input engine), just aren't relevant to Nvim anymore, because we've significantly enhanced and redesigned those subsystems to make them simpler.

UNWANTED VIM-PATCHES

- "channel"

- "job"
- "modelinestrict" https://github.com/neovim/neovim/issues/38985
- "tabpanel"
- "terminal"
- Vim9script features, and anything related to `:scriptversion`. (Nvim supports Vimscript version 1 only.)
- Fixes for build warnings.
- GUI-related changes.
- Most new Vimscript functions or Ex commands (with rare exceptions).

VERSION.C

dev-vimpatch-version.c

The list of Vim patches in `src/nvim/version.c` is automatically updated https://github.com/neovim/neovim/pull/7780 based on the presence of `vim-patch:xxx` tokens in the Nvim git log.

- Don't update `src/nvim/version.c` yourself.
  - `scripts/vim-patch.sh -p` intentionally omits `version.c` to avoid merge conflicts and save time when porting a patch.
- The automation script (`scripts/vimpatch.lua`) only recognizes tokens like `vim-patch:8.0.1206`, not `vim-patch:<hash>`.

CODE DIFFERENCES

dev-vimpatch-diff

The following functions have been removed or deprecated in favor of newer alternatives. See `memory.c` https://github.com/neovim/neovim/blob/master/src/nvim/memory.c for more information.

```
------------------------------------------------------------------------
Deprecated or removed                          Replacement
------------------------------------------------------------------------
vim_free                                            xfree
VIM_CLEAR(&foo)                                XFREE_CLEAR(foo)
malloc alloc lalloc lalloc_id ALLOC_ONE             xmalloc
calloc lalloc_clear                                 xcalloc
realloc vim_realloc                                xrealloc
mch_memmove                                         memmove
vim_memset copy_chars copy_spaces                   memset
vim_strbyte                                         strchr
vim_strncpy strncpy                           xstrlcpy/xmemcpyz
vim_strcat strncat                                  xstrlcat
VIM_ISWHITE                                     ascii_iswhite
IS_WHITE_OR_NUL                               ascii_iswhite_or_nul
IS_WHITE_NL_OR_NUL                           ascii_iswhite_nl_or_nul
vim_isalpha                                       mb_isalpha
vim_isNormalIDc                                 ascii_isident
vim_islower vim_isupper                       mb_islower mb_isupper
vim_tolower vim_toupper                       mb_tolower mb_toupper
mb_ptr2len                                        utfc_ptr2len
mb_ptr2len_len                                  utfc_ptr2len_len
mb_char2len                                       utf_char2len
mb_char2bytes                                    utf_char2bytes
mb_ptr2cells                                      utf_ptr2cells
mb_ptr2cells_len                                utf_ptr2cells_len
mb_char2cells                                    utf_char2cells
mb_off2cells                                      utf_off2cells
mb_ptr2char                                       utf_ptr2char
mb_head_off                                       utf_head_off
mb_tail_off                                       utf_cp_bounds
mb_strnicmp2                                      utf_strnicmp
MB_STRNICMP2                                      utf_strnicmp
mb_lefthalve                                     grid_lefthalve
mb_fix_col                                        grid_fix_col
utf_off2cells                                    grid_off2cells
ml_get_curline                                 get_cursor_line_ptr
ml_get_cursor                                  get_cursor_pos_ptr
ml_get_curline_len                             get_cursor_line_len
ml_get_cursor_len                              get_cursor_pos_len
screen_char                                           ui_line
screen_line                                     grid_put_linebuf
screen_* (most functions)                             grid_*
update_prepare, update_finish #9484              removed; use update_screen only
ARRAY_LENGTH                                      ARRAY_SIZE
vim_strsave_escape_csi                        vim_strsave_escape_ks
vim_unescape_csi                                 vim_unescape_ks
gettail                                             path_tail
mch_isFullName                                  path_is_absolute
script_do_profile                                 profile_init

------------------------------------------------------------------------
```

Make sure to note the difference between `utf_` and `utfc_` when replacing `mb_` functions. Also indirect call syntax `(*mb_ptr2len)(...)` should be replaced with an ordinary function call `utfc_ptr2len(...)`.

```
------------------------------------------------------------------------
Data type    Format (Vim source)     Portable format (Nvim source)
-----------  ----------------------- ---------------------------------
long long    "%lld"                  "%" PRId64
size_t       "%ld"                   "%zu"
linenr_T     "%ld"                   "%" PRIdLINENR
------------------------------------------------------------------------
```

- See also: https://github.com/neovim/neovim/pull/1729#discussion_r22423779
- Vim's `ga_init2` was renamed to `ga_init` and the original `ga_init` is gone.
- "Old style" Vim tests (`src/testdir/*.in`) should be converted to Lua tests (see #1286 https://github.com/neovim/neovim/issues/1286 and #1328 https://github.com/neovim/neovim/pull/1328). See Checklist for migrating legacy tests https://github.com/neovim/neovim/blob/master/test/README.md#checklist-for-migrating-legacy-tests.
  - However, please do not convert "new style" Vim tests (`src/testdir/*.vim`) to Lua. The "new style" Vim tests are faster than the old ones, and converting them takes time and effort better spent elsewhere. Just copy them to `test/old/testdir/*.vim`.
- Conditions that check `enc_utf8` or `has_mbyte` are obsolete (only the "true" case is applicable).
  - `enc_utf8` and `has_mbyte` macros were removed in https://github.com/neovim/neovim/pull/13293
- Check for `CSI` in typeahead buffer is only necessary in Vim with `FEAT_GUI`. `CSI` does not have a special meaning in typeahead buffer in Nvim. (also see https://github.com/neovim/neovim/pull/16936)

LIST MANAGEMENT

dev-vimpatch-list-impl

Management of lists (types `list_T` and `listitem_T` from vim) was changed in https://github.com/neovim/neovim/pull/7708/. There is a lint against the "old" usage, but here are the most important changes.

Declarations for the table

- `list_T list`: a list
- `listitem_T li`: an item of `list`
- `int val` a value for `lv_copyID`

```
--------------------------------------------------------------------------------
Old                           New                                 Comment
----------------------------- ----------------------------------------------------
list->lv_first                tv_list_first(list)
list->lv_last                 tv_list_last(list)
li->li_next                   TV_LIST_ITEM_NEXT(list, li)         To be avoided if possible, must use list which li belongs to.
li->li_prev                   TV_LIST_ITEM_PREV(list, li)         To be avoided if possible, must use list which li belongs to.
                              Suggestion by @ZyX-l:               Use TV_LIST_ITER or indexing instead of the previous two calls.
list->lv_len                  tv_list_len(list)
list->lv_lock                 tv_list_locked(list)
&li->li_tv                    TV_LIST_ITEM_TV(li)
list->lv_refcount++           tv_list_ref(list)
val = list->lv_copyID         val = tv_list_copyid(list)
list->lv_copyID = val         tv_list_set_copyid(list, val)

for (li = list->lv_first;     TV_LIST_ITER_CONST(list, li,        Use TV_LIST_ITER(...) if you need to
li != NULL && another_cond;   { if (another_cond) {break;} code})  modify list items (note: assigning copyID is also modification and
this happens
li = li->li_next) code                                            always when recursively traversing a list).

--------------------------------------------------------------------------------
```

For more details and some more advanced usage, see `typval.h` and `typval.c`.

DOCUMENTATION DIFFERENCES

dev-vimpatch-doc

The following should be removed from all imported documentation, and not be used in new documentation:

- `{Only when compiled with ...}`: the vast majority of features have been made non-optional (see https://github.com/neovim/neovim/wiki/Introduction)

FILETYPE DETECTION

dev-vimpatch-filetype

Nvim's filetype detection behavior matches Vim, but is implemented as part of vim.filetype (see `$VIMRUNTIME/lua/vim/filetype.lua`). The logic is encoded in three tables, listed in order of precedence (the first match is returned):

1. `filename` for literal full path or basename lookup;
2. `pattern` for matching filenames or paths against lua-patterns, optimized for fast lookup;
3. `extension` for literal extension lookup.

Logic that requires checking file contents or buffer variables is implemented in `$VIMRUNTIME/lua/vim/filetype/detect.lua`.

When porting filetype patches from Vim, keep the following in mind:

Prefer explicit filenames or extensions over patterns, especially for case insensitive matches (see https://github.com/neovim/neovim/pull/29800):

```
"*[mM]akefile" regex -> "makefile", "Makefile" filenames
"*.js\c"       regex -> "js", "jS", "Js", "jS" extensions
```

Pattern matching has several differences:

- It is done using explicit Lua patterns without implicit anchoring instead of Vim regexes:

```
"*/debian/changelog" -> "/debian/changelog$"
"*/bind/db.*"        -> "/bind/db%."
```

- Filetype patterns are grouped by their parent pattern to improve matching performance: If the parent pattern does not match, skip testing all child patterns. Note that unlike leaf patterns, parent patterns do not have special matching behaviour if they contain a `/`.

When adding a new filetype with pattern matching, consider the following:

- If there is already a group with appropriate parent pattern, use it.
- If there can be a fast and specific enough pattern to group at least 3 filetype patterns, add it as a separate grouped entry.

New parent patterns should be

- fast: rule of thumb is that it should be a short explicit string (i.e. no quantifiers or character sets);
- specific: rules of thumb, in order:
  - full directory name (e.g., `"/etc/"`, `"/log/"`);
  - part of a rare enough directory name (e.g., `"/conf"`, `"git/"`);
  - string rarely used in real full paths (e.g., `"nginx"`).

Example:

- Filetype pattern: `".*/etc/a2ps/.*%.cfg"`
- Good parents: `"/etc/"` or `"%.cfg$"`
- Bad parents: `"%."` (fast but not specific) or `"/a2ps/.*%."` (specific but slow)

When modifying an existing regular pattern, make sure that it still fits its group.

Diagnostic framework

Nvim provides a framework for displaying errors or warnings from external tools, otherwise known as "diagnostics". These diagnostics can come from a variety of sources, such as linters or LSP servers. The diagnostic framework is an extension to existing error handling functionality such as the quickfix list.

QUICKSTART diagnostic-quickstart

Anything that reports diagnostics is referred to below as a "diagnostic producer". Diagnostic producers need only follow a few simple steps to report diagnostics:

1. Create a namespace nvim_create_namespace(). Note that the namespace must have a name. Anonymous namespaces WILL NOT WORK.
2. (Optional) Configure options for the diagnostic namespace vim.diagnostic.config().
3. Generate diagnostics.
4. Set the diagnostics for the buffer vim.diagnostic.set().
5. Repeat from step 3.

Generally speaking, the API is split between functions meant to be used by diagnostic producers and those meant for diagnostic consumers (i.e. end users who want to read and view the diagnostics for a buffer). The APIs for producers require a `{namespace}` as their first argument, while those for consumers generally do not require a namespace (though often one may be optionally supplied). A good rule of thumb is that if a method is meant to modify the diagnostics for a buffer (e.g. vim.diagnostic.set()) then it requires a namespace.

vim.diagnostic.severity diagnostic-severity The "severity" key in a diagnostic is one of the values defined in `vim.diagnostic.severity`:

```
vim.diagnostic.severity.ERROR
vim.diagnostic.severity.WARN
vim.diagnostic.severity.INFO
vim.diagnostic.severity.HINT
```

Functions that take a severity as an optional parameter (e.g. vim.diagnostic.get()) allow using severity names (like "ERROR") as well as values and accept one of three forms:

1. A single vim.diagnostic.severity value:

```
vim.diagnostic.get(0, { severity = vim.diagnostic.severity.WARN })
```

2. A table with a "min" or "max" key (or both). This form allows users to specify a range of severities:

```
vim.diagnostic.get(0, { severity = { min = vim.diagnostic.severity.WARN } })
```

3. A list-like table. This form allows filtering for specific severities:

```
vim.diagnostic.get(0, { severity = {
    vim.diagnostic.severity.WARN,
    vim.diagnostic.severity.INFO,
} })
```

DEFAULTS diagnostic-defaults

These diagnostic keymaps are created unconditionally when Nvim starts:

-]d jumps to the next diagnostic in the buffer.]d-default
- [d jumps to the previous diagnostic in the buffer. [d-default
-]D jumps to the last diagnostic in the buffer.]D-default
- [D jumps to the first diagnostic in the buffer. [D-default
- <C-w>d shows diagnostic at cursor in a floating window. CTRL-W_d-default

HANDLERS diagnostic-handlers

Diagnostics are shown to the user with vim.diagnostic.show(). The display of diagnostics is managed through handlers. A handler is a table with a "show" and (optionally) a "hide" function. The "show" function has the signature

```
function(namespace, bufnr, diagnostics, opts)
```

and is responsible for displaying or otherwise handling the given diagnostics. The "hide" function takes care of "cleaning up" any actions taken by the "show" function and has the signature

```
function(namespace, bufnr)
```

Handlers can be configured with vim.diagnostic.config() and added by creating a new key in `vim.diagnostic.handlers` (see diagnostic-handlers-example).

The `{opts}` table passed to a handler is the full set of configuration options (that is, it is not limited to just the options for the handler itself). The values in the table are already resolved (i.e. if a user specifies a function for a config option, the function has already been evaluated).

If a diagnostic handler is configured with a "severity" key then the list of diagnostics passed to that handler will be filtered using the value of that key (see example below).

Nvim provides these handlers by default: "virtual_text", "virtual_lines", "signs", and "underline".

diagnostic-handlers-example

The example below creates a new handler that notifies the user of diagnostics with vim.notify():

```
-- It's good practice to namespace custom handlers to avoid collisions
vim.diagnostic.handlers["my/notify"] = {
  show = function(namespace, bufnr, diagnostics, opts)
    -- In our example, the opts table has a "log_level" option
    local level = opts["my/notify"].log_level

    local name = vim.diagnostic.get_namespace(namespace).name
    local msg = string.format("%d diagnostics in buffer %d from %s",
                              #diagnostics,
                              bufnr,
                              name)
    vim.notify(msg, level)
  end,
}

-- Users can configure the handler
vim.diagnostic.config({
  ["my/notify"] = {
    log_level = vim.log.levels.INFO,

    -- This handler will only receive "error" diagnostics.
    severity = vim.diagnostic.severity.ERROR,
  }
})
```

In this example, there is nothing to do when diagnostics are hidden, so we omit the "hide" function.

Existing handlers can be overridden. For example, use the following to only show a sign for the highest severity diagnostic on a given line:

```
-- Create a custom namespace. This will aggregate signs from all other
-- namespaces and only show the one with the highest severity on a
-- given line
local ns = vim.api.nvim_create_namespace("my_namespace")

-- Get a reference to the original signs handler
local orig_signs_handler = vim.diagnostic.handlers.signs

-- Override the built-in signs handler
vim.diagnostic.handlers.signs = {
  show = function(_, bufnr, _, opts)
    -- Get all diagnostics from the whole buffer rather than just the
    -- diagnostics passed to the handler
    local diagnostics = vim.diagnostic.get(bufnr)

    -- Find the "worst" diagnostic per line
    local max_severity_per_line = {}
    for _, d in pairs(diagnostics) do
      local m = max_severity_per_line[d.lnum]
      if not m or d.severity < m.severity then
        max_severity_per_line[d.lnum] = d
      end
    end

    -- Pass the filtered diagnostics (with our custom namespace) to
    -- the original handler
    local filtered_diagnostics = vim.tbl_values(max_severity_per_line)
    orig_signs_handler.show(ns, bufnr, filtered_diagnostics, opts)
  end,
  hide = function(_, bufnr)
    orig_signs_handler.hide(ns, bufnr)
  end,
}
```

diagnostic-toggle-virtual-lines-example

Diagnostic handlers can also be toggled. For example, you might want to toggle the `virtual_lines` handler with the following keymap:

```
vim.keymap.set('n', 'gK', function()
  local new_config = not vim.diagnostic.config().virtual_lines
  vim.diagnostic.config({ virtual_lines = new_config })
end, { desc = 'Toggle diagnostic virtual_lines' })
```

diagnostic-on-jump-example

You can use the `on_jump` option from vim.diagnostic.jump() to show the diagnostic that was jumped to using a specific handler. For example, the following uses the `virtual_lines` handler when jumping to a diagnostic:

```
--- @param diagnostic? vim.Diagnostic
--- @param bufnr integer
local function on_jump(diagnostic, bufnr)
  if not diagnostic then return end

  vim.diagnostic.show(
    diagnostic.namespace,
    bufnr,
    { diagnostic },
    { virtual_lines = { current_line = true }, virtual_text = false }
  )
end

vim.diagnostic.config({ jump = { on_jump = on_jump } })
```

diagnostic-loclist-example

Whenever the location-list is opened, the following show handler will show the most recent diagnostics:

```
vim.diagnostic.handlers.loclist = {
  show = function(_, _, _, opts)
    -- Generally don't want it to open on every update
    opts.loclist.open = opts.loclist.open or false
    local winid = vim.api.nvim_get_current_win()
    vim.diagnostic.setloclist(opts.loclist)
    vim.api.nvim_set_current_win(winid)
  end
}
```

The handler accepts the same options as vim.diagnostic.setloclist() and can be configured using vim.diagnostic.config():

```
-- Open the location list on every diagnostic change (warnings/errors only).
vim.diagnostic.config({
  loclist = {
    open = true,
    severity = { min = vim.diagnostic.severity.WARN },
  }
})
```

HIGHLIGHTS

diagnostic-highlights

All highlights defined for diagnostics begin with `Diagnostic` followed by the type of highlight (e.g., `Sign`, `Underline`, etc.) and the severity (e.g. `Error`, `Warn`, etc.)

By default, highlights for signs, floating windows, and virtual text are linked to the corresponding default highlight. Underline highlights are not linked and use their own default highlight groups.

For example, the default highlighting for hl-DiagnosticSignError is linked to hl-DiagnosticError. To change the default (and therefore the linked highlights), use the :highlight command:

```
highlight DiagnosticError guifg="BrightRed"
```

hl-DiagnosticError

DiagnosticError Used as the base highlight group. Other Diagnostic highlights link to this by default (except Underline)

hl-DiagnosticWarn

DiagnosticWarn Used as the base highlight group. Other Diagnostic highlights link to this by default (except Underline)

hl-DiagnosticInfo

DiagnosticInfo Used as the base highlight group. Other Diagnostic highlights link to this by default (except Underline)

hl-DiagnosticHint

DiagnosticHint Used as the base highlight group. Other Diagnostic highlights link to this by default (except Underline)

hl-DiagnosticOk

DiagnosticOk Used as the base highlight group. Other Diagnostic highlights link to this by default (except Underline)

hl-DiagnosticVirtualTextError

DiagnosticVirtualTextError Used for "Error" diagnostic virtual text.

hl-DiagnosticVirtualTextWarn

DiagnosticVirtualTextWarn Used for "Warn" diagnostic virtual text.

hl-DiagnosticVirtualTextInfo

DiagnosticVirtualTextInfo Used for "Info" diagnostic virtual text.

hl-DiagnosticVirtualTextHint

DiagnosticVirtualTextHint Used for "Hint" diagnostic virtual text.

hl-DiagnosticVirtualTextOk

DiagnosticVirtualTextOk Used for "Ok" diagnostic virtual text.

hl-DiagnosticVirtualLinesError

DiagnosticVirtualLinesError Used for "Error" diagnostic virtual lines.

hl-DiagnosticVirtualLinesWarn

DiagnosticVirtualLinesWarn Used for "Warn" diagnostic virtual lines.

hl-DiagnosticVirtualLinesInfo

DiagnosticVirtualLinesInfo Used for "Info" diagnostic virtual lines.

hl-DiagnosticVirtualLinesHint

DiagnosticVirtualLinesHint Used for "Hint" diagnostic virtual lines.

hl-DiagnosticVirtualLinesOk

DiagnosticVirtualLinesOk Used for "Ok" diagnostic virtual lines.

hl-DiagnosticUnderlineError

DiagnosticUnderlineError Used to underline "Error" diagnostics.

hl-DiagnosticUnderlineWarn

DiagnosticUnderlineWarn Used to underline "Warn" diagnostics.

hl-DiagnosticUnderlineInfo

DiagnosticUnderlineInfo Used to underline "Info" diagnostics.

hl-DiagnosticUnderlineHint

DiagnosticUnderlineHint Used to underline "Hint" diagnostics.

hl-DiagnosticUnderlineOk

DiagnosticUnderlineOk Used to underline "Ok" diagnostics.

hl-DiagnosticFloatingError

DiagnosticFloatingError Used to color "Error" diagnostic messages in diagnostics float. See vim.diagnostic.open_float()

hl-DiagnosticFloatingWarn

DiagnosticFloatingWarn Used to color "Warn" diagnostic messages in diagnostics float.

hl-DiagnosticFloatingInfo

DiagnosticFloatingInfo Used to color "Info" diagnostic messages in diagnostics float.

hl-DiagnosticFloatingHint

DiagnosticFloatingHint Used to color "Hint" diagnostic messages in diagnostics float.

hl-DiagnosticFloatingOk

DiagnosticFloatingOk Used to color "Ok" diagnostic messages in diagnostics float.

hl-DiagnosticSignError

DiagnosticSignError Used for "Error" signs in sign column.

hl-DiagnosticSignWarn

DiagnosticSignWarn Used for "Warn" signs in sign column.

hl-DiagnosticSignInfo

DiagnosticSignInfo Used for "Info" signs in sign column.

hl-DiagnosticSignHint

DiagnosticSignHint Used for "Hint" signs in sign column.

hl-DiagnosticSignOk

DiagnosticSignOk Used for "Ok" signs in sign column.

hl-DiagnosticDeprecated

DiagnosticDeprecated Used for deprecated or obsolete code. Applied by the "underline" handler; disabled when `vim.diagnostic.config({ underline = false })`.

hl-DiagnosticUnnecessary

DiagnosticUnnecessary Used for unnecessary or unused code. Applied by the "underline" handler; disabled when `vim.diagnostic.config({ underline = false })`.

SIGNS

diagnostic-signs

Signs are defined for each diagnostic severity. The default text for each sign is the first letter of the severity name (for example, "E" for ERROR). Signs can be customized with vim.diagnostic.config(). Example:

```
-- Highlight entire line for errors
-- Highlight the line number for warnings
vim.diagnostic.config({
    signs = {
        text = {
            [vim.diagnostic.severity.ERROR] = '',
            [vim.diagnostic.severity.WARN] = '',
        },
        linehl = {
            [vim.diagnostic.severity.ERROR] = 'ErrorMsg',
        },
        numhl = {
            [vim.diagnostic.severity.WARN] = 'WarningMsg',
        },
    },
})
```

When the "severity_sort" option is set (see vim.diagnostic.config()) the priority of each sign depends on the severity of the associated diagnostic. Otherwise, all signs have the same priority (the value of the "priority" option in the "signs" table of vim.diagnostic.config() or 10 if unset).

EVENTS

diagnostic-events

DiagnosticChanged

DiagnosticChanged After diagnostics have changed. When used from Lua, the new diagnostics are passed to the autocmd callback in the "data" table. Triggered per buffer.

Example:

```
vim.api.nvim_create_autocmd('DiagnosticChanged', {
  callback = function(ev)
    local diagnostics = ev.data.diagnostics
    vim.print(diagnostics)
  end,
})
```

Lua module: vim.diagnostic

diagnostic-api

vim.Diagnostic Extends: vim.Diagnostic.Set

diagnostic-structure

Diagnostics use api-indexing (i.e. 0-based rows, and columns given as 0-based byte indices). See also vim.pos and vim.range to convert positions from other systems.

Fields:
- {bufnr} (integer) Buffer number
- {col} (integer) The starting column of the diagnostic (0-indexed)
- {end_col} (integer) The final column of the diagnostic (0-indexed)
- {end_lnum} (integer) The final line of the diagnostic (0-indexed)
- {namespace}? (integer)
- {severity} (vim.diagnostic.Severity) The severity of the diagnostic vim.diagnostic.severity

vim.Diagnostic.Set Diagnostics use the same indexing as the rest of the Nvim API (i.e. 0-based rows, and columns given as 0-based byte indices). api-indexing

Fields:
- {code}? (string|integer) The diagnostic code
- {col}? (integer, default: 0) The starting column of the diagnostic (0-indexed)
- {end_col}? (integer, default: col) The final column of the diagnostic (0-indexed)
- {end_lnum}? (integer, default: lnum) The final line of the diagnostic (0-indexed)
- {lnum} (integer) The starting line of the diagnostic (0-indexed)
- {message} (string) The diagnostic text
- {severity}? (vim.diagnostic.Severity, default: vim.diagnostic.severity.ERROR) The severity of the diagnostic vim.diagnostic.severity
- {source}? (string) The source of the diagnostic
- {user_data}? (any) arbitrary data plugins can add

vim.diagnostic.GetOpts A table with the following keys:

Fields:
- {enabled}? (boolean, default: nil) Limit diagnostics to only enabled or disabled. If nil, enablement is ignored. See vim.diagnostic.enable()
- {lnum}? (integer) Limit diagnostics to those spanning the specified line number (0-indexed, see diagnostic-structure).
- {namespace}? (integer[]|integer) Limit diagnostics to one or more namespaces.
- {severity}? (vim.diagnostic.SeverityFilter) See diagnostic-severity.

vim.diagnostic.JumpOpts Extends: vim.diagnostic.GetOpts

Configuration table with the keys listed below. Some parameters can have their default values changed with vim.diagnostic.config().

Fields:
- {count}? (integer) The number of diagnostics to move by, starting from {pos}. A positive integer moves forward by {count} diagnostics, while a negative integer moves backward by {count} diagnostics. Mutually exclusive with {diagnostic}.
- {diagnostic}? (vim.Diagnostic) The diagnostic to jump to. Mutually exclusive with {count}, {namespace}, and {severity}.
- {on_jump}? (fun(diagnostic:vim.Diagnostic?, bufnr:integer)) Optional callback invoked with the diagnostic that was jumped to.
- {pos}? ([integer,integer]) Cursor position as a (row, col) tuple. See nvim_win_get_cursor(). Used to find the nearest diagnostic when {count} is used. Only used when {count} is non-nil. Default is the current cursor position.
- {severity}? (vim.diagnostic.SeverityFilter) See diagnostic-severity.
- {winid}? (integer, default: 0) Window ID
- {wrap}? (boolean, default: true) Whether to loop around file or not. Similar to 'wrapscan'.

vim.diagnostic.NS

Fields:
- {disabled}? (boolean)
- {name} (string)
- {opts} (vim.diagnostic.Opts)
- {user_data} (table)

vim.diagnostic.Opts Many of the configuration options below accept one of the following:
- false: Disable this feature
- true: Enable this feature, use default settings.
- table: Enable this feature with overrides. Use an empty table to use default values.
- function: Function with signature (namespace, bufnr) that returns any of the above.

Fields:

- {float}? (boolean|vim.diagnostic.Opts.Float|fun(namespace?: integer|integer[], bufnr:integer): vim.diagnostic.Opts.Float) Options for floating windows. See vim.diagnostic.Opts.Float.
- {jump}? (vim.diagnostic.Opts.Jump) Default values for vim.diagnostic.jump(). See vim.diagnostic.Opts.Jump.
- {severity_sort}? (boolean|{reverse?:boolean}, default: false) Sort diagnostics by severity. This affects the order in which signs, virtual text, and highlights are displayed. When true, higher severities are displayed before lower severities (e.g. ERROR is displayed before WARN). Options:
  - {reverse}? (boolean) Reverse sort order
- {signs}? (boolean|vim.diagnostic.Opts.Signs|fun(namespace: integer, bufnr:integer): vim.diagnostic.Opts.Signs, default: true) Use signs for diagnostics diagnostic-signs.
- {status}? (vim.diagnostic.Opts.Status) Options for the statusline component.
- {underline}? (boolean|vim.diagnostic.Opts.Underline|fun(namespace: integer, bufnr:integer): vim.diagnostic.Opts.Underline, default: true) Used to call attention to a diagnostic ("underline" is a misnomer). Controls the hl-DiagnosticUnnecessary and hl-DiagnosticDeprecated highlights.
- {update_in_insert}? (boolean, default: false) Update diagnostics in Insert mode (if false, diagnostics are updated on InsertLeave)
- {virtual_lines}? (boolean|vim.diagnostic.Opts.VirtualLines|fun(namespace: integer, bufnr:integer): vim.diagnostic.Opts.VirtualLines, default: false) Use virtual lines for diagnostics.
- {virtual_text}? (boolean|vim.diagnostic.Opts.VirtualText|fun(namespace: integer, bufnr:integer): vim.diagnostic.Opts.VirtualText, default: false) Use virtual text for diagnostics. If multiple diagnostics are set for a namespace, one prefix per diagnostic + the last diagnostic message are shown.

vim.diagnostic.Opts.Float Extends: vim.lsp.util.open_floating_preview.Opts

Fields:
- {bufnr}? (integer, default: current buffer) Buffer number to show diagnostics from.
- {format}? (fun(diagnostic:vim.Diagnostic): string?) A function that takes a diagnostic as input and returns a string or nil. If the return value is nil, the diagnostic is not displayed by the handler. Else the output text is used to display the diagnostic. Overrides the setting from vim.diagnostic.config().
- {header}? (string|[string,any]) String to use as the header for the floating window. If a table, it is interpreted as a [text, hl_group] tuple. Overrides the setting from vim.diagnostic.config().
- {namespace}? (integer|integer[]) Limit diagnostics to the given namespace(s).
- {pos}? (integer|[integer,integer]) If {scope} is "line" or "cursor", use this position rather than the cursor position. If a number, interpreted as a line number; otherwise, a (row, col) tuple.
- {prefix}? (string|table|(fun(diagnostic:vim.Diagnostic,i:integer,total:integer): string, string)) Prefix each diagnostic in the floating window:
  - If a function, {i} is the index of the diagnostic being evaluated and {total} is the total number of diagnostics displayed in the window. The function should return a string which is prepended to each diagnostic in the window as well as an (optional) highlight group which will be used to highlight the prefix.
  - If a table, it is interpreted as a [text, hl_group] tuple as in nvim_echo()
  - If a string, it is prepended to each diagnostic in the window with no highlight. Overrides the setting from vim.diagnostic.config().
- {scope}? ('line'|'buffer'|'cursor'|'c'|'l'|'b', default: line) Show diagnostics from the whole buffer (buffer), the current cursor line (line), or the current cursor position (cursor). Shorthand versions are also accepted (c for cursor, l for line, b for buffer).
- {severity}? (vim.diagnostic.SeverityFilter) See diagnostic-severity. Overrides the setting from vim.diagnostic.config().
- {severity_sort}? (boolean|{reverse?:boolean}, default: false) Sort diagnostics by severity. Overrides the setting from vim.diagnostic.config().
- {source}? (boolean|'if_many') Include the diagnostic source in the message. Use "if_many" to only show sources if there is more than one source of diagnostics in the buffer. Otherwise, any truthy value means to always show the diagnostic source. Overrides the setting from vim.diagnostic.config().
- {suffix}? (string|table|(fun(diagnostic:vim.Diagnostic,i:integer,total:integer): string, string)) Same as {prefix}, but appends the text to the diagnostic instead of prepending it. Overrides the setting from vim.diagnostic.config().

vim.diagnostic.Opts.Jump

Fields:
- {on_jump}? (fun(diagnostic:vim.Diagnostic?, bufnr:integer)) Default value of the {on_jump} parameter of vim.diagnostic.jump().
- {severity}? (vim.diagnostic.SeverityFilter) Default value of the {severity} parameter of vim.diagnostic.jump().
- {wrap}? (boolean, default: true) Default value of the {wrap} parameter of vim.diagnostic.jump().

vim.diagnostic.Opts.Signs

Fields:
- {linehl}? (table<vim.diagnostic.Severity,string>) A table mapping diagnostic-severity to the highlight group used for the whole line the sign is placed in.
- {numhl}? (table<vim.diagnostic.Severity,string>) A table mapping diagnostic-severity to the highlight group used for the line number where the sign is placed.
- {priority}? (integer, default: 10) Base priority to use for signs. When {severity_sort} is used, the priority of a sign is adjusted based on its severity. Otherwise, all signs use the same priority.
- {severity}? (vim.diagnostic.SeverityFilter) Only show signs for diagnostics matching the given severity diagnostic-severity
- {text}? (table<vim.diagnostic.Severity|vim.diagnostic.SeverityName,string>) A table mapping diagnostic-severity to the sign text to display in the sign column and statusline. The default is to use "E", "W", "I", and "H" for errors, warnings, information, and hints, respectively. Example:

```
vim.diagnostic.config({
  signs = { text = { [vim.diagnostic.severity.ERROR] = 'E', ... } }
})
```

vim.diagnostic.Opts.Status

Fields:
- {format}? (fun(counts:table<vim.diagnostic.Severity,integer>): string) Function that accepts a mapping of diagnostic-severity to the number of diagnostics of the corresponding severity (only those having at least 1 diagnostic) and returns a 'statusline' component. Highlights must be applied by the format function. Example:

```
local signs = {
  [vim.diagnostic.severity.ERROR] = "A",
  -- ...
}
local hl_map = {
  [vim.diagnostic.severity.ERROR] = 'DiagnosticSignError',
  -- ...
}
vim.diagnostic.config({
  status = {
    format = function(severity_counts)
      local items = {}
      for severity in ipairs(vim.diagnostic.severity) do
        local count = severity_counts[severity] or 0
        table.insert(items, ("%%#%s#%s %s"):format(hl_map[severity], signs[severity], count))
      end
      return table.concat(items, " ")
    end
  }
})
```

vim.diagnostic.Opts.Underline

Fields:
- {severity}? (vim.diagnostic.SeverityFilter) Only underline diagnostics matching the given severity diagnostic-severity.

vim.diagnostic.Opts.VirtualLines

Fields:
- {current_line}? (boolean, default: false) Only show diagnostics for the current line. Updated on CursorHold, see 'updatetime'.
- {format}? (fun(diagnostic:vim.Diagnostic): string?) A function that takes a diagnostic as input and returns a string or nil. If the return value is nil, the diagnostic is not displayed by the handler. Else the output text is used to display the diagnostic.
- {overflow}? ('trunc'|'scroll'|'wrap'|'auto', default: auto) See virt_lines_overflow in nvim_buf_set_extmark().
- {severity}? (vim.diagnostic.SeverityFilter) Only show virtual lines for diagnostics matching the given severity diagnostic-severity

vim.diagnostic.Opts.VirtualText

Fields:
- {current_line}? (boolean, default: nil) Show or hide diagnostics based on the current cursor line. If true, only diagnostics on the current cursor line are shown. If false, all diagnostics are shown except on the current cursor line. If nil, all diagnostics are shown. Updated on CursorHold, see 'updatetime'.
- {format}? (fun(diagnostic:vim.Diagnostic): string?) If not nil, the return value is the text used to display the diagnostic. Example:

```
function(diagnostic)
  if diagnostic.severity == vim.diagnostic.severity.ERROR then
    return string.format("E: %s", diagnostic.message)
  end
  return diagnostic.message
end
```

If the return value is nil, the diagnostic is not displayed by the handler.
- {hl_mode}? ('replace'|'combine'|'blend') See nvim_buf_set_extmark().
- {prefix}? (string|(fun(diagnostic:vim.Diagnostic,i:integer,total:integer): string)) Prepend diagnostic message with prefix. If a function, {i} is the index of the diagnostic being evaluated, and {total} is the total number of diagnostics for the line. This can be used to render diagnostic symbols or error codes.
- {severity}? (vim.diagnostic.SeverityFilter) Only show virtual text for diagnostics matching the given severity diagnostic-severity
- {source}? (boolean|"if_many") Include the diagnostic source in virtual text. Use 'if_many' to only show sources if there is more than one diagnostic source in the buffer. Otherwise, any truthy value means to always show the diagnostic source.
- {spacing}? (integer) Amount of empty spaces inserted at the beginning of the virtual text.
- {suffix}? (string|(fun(diagnostic:vim.Diagnostic): string)) Append diagnostic message with suffix. This can be used to render an LSP diagnostic error code.
- {virt_text}? ([string,any][]) See nvim_buf_set_extmark().
- {virt_text_hide}? (boolean) See nvim_buf_set_extmark().
- {virt_text_pos}? ('eol'|'eol_right_align'|'inline'|'overlay'|'right_align') See nvim_buf_set_extmark().
- {virt_text_win_col}? (integer) See nvim_buf_set_extmark().

config({opts}, {namespace}) vim.diagnostic.config()
Configure diagnostic options globally or for a specific diagnostic namespace.

Configuration can be specified globally, per-namespace, or ephemerally (i.e. only for a single call to vim.diagnostic.set() or vim.diagnostic.show()). Ephemeral configuration has highest priority, followed by namespace configuration, and finally global configuration.

For example, if a user enables virtual text globally with

```
vim.diagnostic.config({ virtual_text = true })
```

and a diagnostic producer sets diagnostics with

```
vim.diagnostic.set(ns, 0, diagnostics, { virtual_text = false })
```

then virtual text will not be enabled for those diagnostics.

Parameters:
- {opts} (vim.diagnostic.Opts?) When omitted or nil, retrieve the current configuration. Otherwise, a configuration table (see vim.diagnostic.Opts).
- {namespace} (integer?) Update the options for the given namespace. When omitted, update the global diagnostic options.

Overloads:
- `fun(opts?: nil, namespace?: integer): vim.diagnostic.Opts`

Return:
(`vim.diagnostic.Opts?`) Current diagnostic config if `{opts}` is omitted.

count({buf}, {opts}) vim.diagnostic.count()
Get current diagnostics count.

Parameters:
- `{buf}` (`integer?`) Buffer number to get diagnostics from. Use 0 for current buffer or nil for all buffers.
- `{opts}` (`vim.diagnostic.GetOpts?`)

Return:
(`table<integer, integer>`) Table with actually present severity values as keys (see diagnostic-severity) and integer counts as values.

enable({enable}, {filter}) vim.diagnostic.enable()
Enables or disables diagnostics.

To "toggle", pass the inverse of `is_enabled()`:

```
vim.diagnostic.enable(not vim.diagnostic.is_enabled())
```

Parameters:
- `{enable}` (`boolean?`) true/nil to enable, false to disable
- `{filter}` (`table?`) Optional filters kwargs, or `nil` for all.
  - `{bufnr}?` (`integer`) Buffer number, or 0 for current buffer, or `nil` for all buffers.
  - `{ns_id}?` (`integer`) Diagnostic namespace, or `nil` for all.

fromqflist({list}, {opts}) vim.diagnostic.fromqflist()
Convert a list of quickfix items to a list of diagnostics.

Parameters:
- `{list}` (`vim.fn.getqflist.ret.item[]`) List of quickfix items from getqflist() or getloclist().
- `{opts}` (`table?`) Configuration table with the following keys:
  - `{merge_lines}?` (`boolean`) When true, items with valid=0 are appended to the previous valid item's message with a newline. (default: false)

Return:
(`vim.Diagnostic[]`)

get({buf}, {opts}) vim.diagnostic.get()
Get current diagnostics.

Modifying diagnostics in the returned table has no effect. To set diagnostics in a buffer, use vim.diagnostic.set().

Parameters:
- `{buf}` (`integer?`) Buffer number to get diagnostics from. Use 0 for current buffer or nil for all buffers.
- `{opts}` (`vim.diagnostic.GetOpts?`)

Return:
(`vim.Diagnostic[]`)

get_namespace({namespace}) vim.diagnostic.get_namespace()
Get namespace metadata.

Parameters:
- `{namespace}` (`integer`) Diagnostic namespace

Return:
(`vim.diagnostic.NS`) Namespace metadata

get_namespaces() vim.diagnostic.get_namespaces()
Get current diagnostic namespaces.

Return:
(`table<integer,vim.diagnostic.NS>`) List of active diagnostic namespaces vim.diagnostic.

get_next({opts}) vim.diagnostic.get_next()
Get the next diagnostic closest to the cursor position.

Parameters:
- `{opts}` (`vim.diagnostic.JumpOpts?`)

Return:
(`vim.Diagnostic?`) Next diagnostic

get_prev({opts}) vim.diagnostic.get_prev()
Get the previous diagnostic closest to the cursor position.

Parameters:
- `{opts}` (`vim.diagnostic.JumpOpts?`)

Return:
(`vim.Diagnostic?`) Previous diagnostic

hide({namespace}, {buf}) vim.diagnostic.hide()
Hide currently displayed diagnostics.

This only clears the decorations displayed in the buffer. Diagnostics can be redisplayed with vim.diagnostic.show(). To completely remove diagnostics, use vim.diagnostic.reset().

To hide diagnostics and prevent them from re-displaying, use vim.diagnostic.enable().

Parameters:
- {namespace} (integer?) Diagnostic namespace. When omitted, hide diagnostics from all namespaces.
- {buf} (integer?) Buffer number, or 0 for current buffer. When omitted, hide diagnostics in all buffers.

is_enabled({filter}) vim.diagnostic.is_enabled()
Check whether diagnostics are enabled.

Attributes:
Since: 0.10.0

Parameters:
- {filter} (table?) Optional filters kwargs, or nil for all.
  - {bufnr}? (integer) Buffer number, or 0 for current buffer, or nil for all buffers.
  - {ns_id}? (integer) Diagnostic namespace, or nil for all.

Return:
(boolean)

jump({opts}) vim.diagnostic.jump()
Move to a diagnostic.

Parameters:
- {opts} (vim.diagnostic.JumpOpts)

Return:
(vim.Diagnostic?) The diagnostic that was moved to.

vim.diagnostic.match()

match({str}, {pat}, {groups}, {severity_map}, {defaults}) Parse a diagnostic from a string.

For example, consider a line of output from a linter:

```
WARNING filename:27:3: Variable 'foo' does not exist
```

This can be parsed into a vim.Diagnostic.Set structure with:

```
local s = "WARNING filename:27:3: Variable 'foo' does not exist"
local pattern = "^(%w+) %w+:(%d+):(%d+): (.+)$"
local groups = { "severity", "lnum", "col", "message" }
vim.diagnostic.match(s, pattern, groups, { WARNING = vim.diagnostic.WARN })
```

Parameters:
- {str} (string) String to parse diagnostics from.
- {pat} (string) Lua pattern with capture groups.
- {groups} (string[]) List of fields in a vim.Diagnostic.Set structure to associate with captures from {pat}.
- {severity_map} (table) A table mapping the severity field from {groups} with an item from vim.diagnostic.severity.
- {defaults} (table?) Table of default values for any fields not listed in {groups}. When omitted, numeric values default to 0 and "severity" defaults to ERROR.

Return:
(vim.Diagnostic.Set?) vim.Diagnostic.Set structure or nil if {pat} fails to match {str}.

open_float({opts}) vim.diagnostic.open_float()
Show diagnostics in a floating window.

Parameters:
- {opts} (vim.diagnostic.Opts.Float?)

Return (multiple):
(integer?) float_bufnr (integer?) winid

reset({namespace}, {buf}) vim.diagnostic.reset()
Remove all diagnostics from the given namespace.

Unlike vim.diagnostic.hide(), this function removes all saved diagnostics. They cannot be redisplayed using vim.diagnostic.show(). To simply remove diagnostic decorations in a way that they can be re-displayed, use vim.diagnostic.hide().

Parameters:
- {namespace} (integer?) Diagnostic namespace. When omitted, remove diagnostics from all namespaces.
- {buf} (integer?) Remove diagnostics for the given buffer. When omitted, diagnostics are removed for all buffers.

set({namespace}, {buf}, {diagnostics}, {opts}) vim.diagnostic.set()
Set diagnostics for the given namespace and buffer.

Parameters:
- {namespace} (integer) The diagnostic namespace
- {buf} (integer) Buffer number
- {diagnostics} (vim.Diagnostic.Set[])
- {opts} (vim.diagnostic.Opts?) Display options to pass to vim.diagnostic.show()

setloclist({opts}) vim.diagnostic.setloclist()
Add buffer diagnostics to the location list.

Parameters:
- {opts} (table?) Configuration table with the following keys:
  - {format}? (fun(diagnostic:vim.Diagnostic): string?) A function that takes a diagnostic as input and returns a string or nil. If the return value is nil, the diagnostic is not displayed in the location list. Else the output text is used to display the diagnostic.
  - {namespace}? (integer[]|integer) Only add diagnostics from the given namespace(s).
  - {open}? (boolean, default: true) Open the location list after setting.

- {severity}? (vim.diagnostic.SeverityFilter) See diagnostic-severity.
- {title}? (string) Title of the location list. Defaults to "Diagnostics".
- {winnr}? (integer, default: 0) Window number to set location list for.

setqflist({opts}) vim.diagnostic.setqflist()
Add all diagnostics to the quickfix list.

Parameters:
- {opts} (table?) Configuration table with the following keys:
  - {format}? (fun(diagnostic:vim.Diagnostic): string?) A function that takes a diagnostic as input and returns a string or nil. If the return value is nil, the diagnostic is not displayed in the quickfix list. Else the output text is used to display the diagnostic.
  - {namespace}? (integer[]|integer) Only add diagnostics from the given namespace(s).
  - {open}? (boolean, default: true) Open quickfix list after setting.
  - {severity}? (vim.diagnostic.SeverityFilter) See diagnostic-severity.
  - {title}? (string) Title of quickfix list. Defaults to "Diagnostics". If there's already a quickfix list with this title, it's updated. If not, a new quickfix list is created.

show({namespace}, {buf}, {diagnostics}, {opts}) vim.diagnostic.show() Display diagnostics for the given namespace and buffer.

Parameters:
- {namespace} (integer?) Diagnostic namespace. When omitted, show diagnostics from all namespaces.
- {buf} (integer?) Buffer number, or 0 for current buffer. When omitted, show diagnostics in all buffers.
- {diagnostics} (vim.Diagnostic[]?) The diagnostics to display. When omitted, use the saved diagnostics for the given namespace and buffer. This can be used to display a list of diagnostics without saving them or to display only a subset of diagnostics. May not be used when {namespace} or {buf} is nil.
- {opts} (vim.diagnostic.Opts?) Display options.

status({buf}) vim.diagnostic.status()
Returns formatted string with diagnostics for the current buffer. The severities with 0 diagnostics are left out. Example E:2 W:3 I:4 H:5

To customise appearance, see vim.diagnostic.Opts.Status.

Parameters:
- {buf} (integer?) Buffer number to get diagnostics from. Defaults to 0 for the current buffer

Return:
(string)

toqflist({diagnostics}) vim.diagnostic.toqflist()
Convert a list of diagnostics to a list of quickfix items that can be passed to setqflist() or setloclist().

Parameters:
- {diagnostics} (vim.Diagnostic[])

Return:
(table[]) Quickfix list items setqflist-what

diff-mode
This file describes the diff feature: Showing differences between two to
eight versions of the same file.

The basics are explained in section 08.7 of the user manual.

1. Starting diff mode start-vimdiff

To start editing in diff mode, run "nvim -d". This starts Nvim as usual, and
additionally sets up for viewing the differences between the arguments.

```
nvim -d file1 file2 [file3 [file4]]
```

In addition to the -d argument, -R may be used for readonly mode.

The second and following arguments may also be a directory name. Vim will
then append the file name of the first argument to the directory name to find
the file.

By default an internal diff library will be used. When 'diffopt' or
'diffexpr' has been set an external "diff" command will be used. This only
works when such a diff program is available.

Diffs are local to the current tabpage. You can't see diffs with a window
in another tabpage. This does make it possible to have several diffs at the
same time, each in their own tabpage.

What happens is that Nvim opens a window for each of the files. This is like
using the -O argument. This uses vertical splits, but if you prefer
horizontal splits use the -o argument instead:

```
nvim -d -o file1 file2 [file3 [file4]]
```

If you always prefer horizontal splits include "horizontal" in 'diffopt'.

In each of the edited files these options are set:

'diff'	on
'scrollbind'	on
'cursorbind'	on
'scrollopt'	includes "hor"
'wrap'	off, or leave as-is if 'diffopt' includes "followwrap"
'foldmethod'	"diff"
'foldcolumn'	value from 'diffopt', default is 2

These options are set local to the window. When editing another file they are
reset to the global value.
The options can still be overruled from a modeline when re-editing the file.
However, 'foldmethod' and 'wrap' won't be set from a modeline when 'diff' is
set.
See :diffoff for an easy way to revert the options.

The differences shown are actually the differences in the buffer. Thus if you
make changes after loading a file, these will be included in the displayed
diffs. You might have to do ":diffupdate" now and then, not all changes are
immediately taken into account, especially when using an external diff
command.

In your vimrc file you could do something special when Vim was started in
diff mode. You could use a construct like this:

```
if &diff
   setup for diff mode
else
   setup for non-diff mode
endif
```

While already in Vim you can start diff mode in three ways.

E98

:diffs[plit] {filename} :diffs :diffsplit
Open a new window on the file {filename}. The options are set
as for "nvim -d" for the current and the newly opened window.
Also see 'diffexpr'.

:difft[his] :difft :diffthis
Make the current window part of the diff windows. This sets
the options as for "nvim -d".

:diffp[atch] {patchfile} E816 :diffp :diffpatch
Use the current buffer, patch it with the diff found in
{patchfile} and open a buffer on the result. This sets the
options as for "nvim -d".
{patchfile} can be in any format that the "patch" program
understands or 'patchexpr' can handle.
Note that {patchfile} should only contain a diff for one file,
the current file. If {patchfile} contains diffs for other
files as well, the results are unpredictable. Vim changes
directory to /tmp to avoid files in the current directory
accidentally being patched. But it may still result in
various ".rej" files to be created. And when absolute path
names are present these files may get patched anyway.

To make these commands use a vertical split, prepend :vertical. Examples:

```
:vert diffsplit main.c~
:vert diffpatch /tmp/diff
```

If you always prefer a vertical split include "vertical" in 'diffopt'.

E96

There can be up to eight buffers with 'diff' set.

Since the option values are remembered with the buffer, you can edit another file for a moment and come back to the same file and be in diff mode again.

:diffo :diffoff

:diffo[ff] Switch off diff mode for the current window. Resets related options also when 'diff' was not set.

:diffo[ff]! Switch off diff mode for the current window and in all windows in the current tabpage where 'diff' is set. Resetting related options only happens in a window that has 'diff' set, if the current window does not have 'diff' set then no options in it are changed.
Hidden buffers are also removed from the list of diff'ed buffers.

The :diffoff command resets the relevant options to the values they had when using :diffsplit, :diffpatch, :diffthis, or starting Vim in diff mode.
When using :diffoff twice the last saved values are restored.
Otherwise they are set to their default value:

'diff'	off
'scrollbind'	off
'cursorbind'	off
'scrollopt'	without "hor"
'wrap'	on, or leave as-is if 'diffopt' includes "followwrap"
'foldmethod'	"manual"
'foldcolumn'	0

'foldenable' will most-likely be reset to off. That is when 'foldmethod' is restored to "manual". The folds themselves are not cleared but they should not show up, resetting 'foldenable' is the best way to do that.

2. Viewing diffs view-diffs

The effect is that the diff windows show the same text, with the differences highlighted. When scrolling the text, the 'scrollbind' option will make the text in other windows to be scrolled as well. With vertical splits the text should be aligned properly.

The alignment of text will go wrong when:

- 'wrap' is on, some lines will be wrapped and occupy two or more screen lines
- folds are open in one window but not another
- 'scrollbind' is off
- changes have been made to the text
- "filler" is not present in 'diffopt', deleted/inserted lines makes the alignment go wrong

All the buffers edited in a window where the 'diff' option is set will join in the diff. This is also possible for hidden buffers. They must have been edited in a window first for this to be possible. To get rid of the hidden buffers use :diffoff!.

:DiffOrig diff-original-file

Since 'diff' is a window-local option, it's possible to view the same buffer in diff mode in one window and "normal" in another window. It is also possible to view the changes you have made to a buffer since the file was loaded. Since Vim doesn't allow having two buffers for the same file, you need another buffer. This command is useful:

```
command DiffOrig vert new | set buftype=nofile | read ++edit # | 0d_
       \ | diffthis | wincmd p | diffthis
```

Then use ":DiffOrig" to see the differences
between the current buffer and the file it was loaded from.

A buffer that is unloaded cannot be used for the diff. But it does work for hidden buffers. You can use ":hide" to close a window without unloading the buffer. If you don't want a buffer to remain used for the diff do ":set nodiff" before hiding it.

:dif :diff :diffupdate

:dif[fupdate][!] Update the diff highlighting and folds.

Vim attempts to keep the differences updated when you make changes to the text. This mostly takes care of inserted and deleted lines. Changes within a line and more complicated changes do not cause the differences to be updated. To force the differences to be updated use:

```
:diffupdate
```

If the ! is included Vim will check if the file was changed externally and needs to be reloaded. It will prompt for each changed file, like :checktime was used.

Vim will show filler lines for lines that are missing in one window but are present in another. These lines were inserted in another file or deleted in this file. Removing "filler" from the 'diffopt' option will make Vim not display these filler lines.

Folds are used to hide the text that wasn't changed. See folding for all the commands that can be used with folds.

The context of lines above a difference that are not included in the fold can be set with the 'diffopt' option. For example, to set the context to three lines:

```
:set diffopt=filler,context:3
```

The diffs are highlighted with these groups:

hl-DiffAdd	DiffAdd	Added (inserted) lines. These lines exist in this buffer but not in another.
hl-DiffChange	DiffChange	Changed lines.
hl-DiffText	DiffText	Changed text inside a Changed line. Exact behavior depends on the inline: setting in 'diffopt'. With inline: set to "simple", Vim finds the first character that is different, and the last character that is different (searching from the end of the line). The text in between is highlighted. This means that parts in the middle that are still the same are highlighted anyway. The 'diffopt' flags "iwhite" and "icase" are used here. With inline: set to "char" or "word", Vim uses the internal diff library to perform a detailed diff between the changed blocks and highlight the exact difference between the two. Will respect any 'diffopt' flag that affects internal diff. Not used when inline: is set to "none".
hl-DiffTextAdd	DiffTextAdd	Added text inside a Changed line. Similar to DiffText, but used when there is no corresponding text in other buffers. Not used when inline: is set to "simple" or "none".
hl-DiffDelete	DiffDelete	Deleted lines. Also called filler lines, because they don't really exist in this buffer.

3. Jumping to diffs jumpto-diffs

Two commands can be used to jump to diffs:

[c

```
[c              Jump backwards to the previous start of a change.
                When a count is used, do it that many times.
```

]c

```
]c              Jump forwards to the next start of a change.
                When a count is used, do it that many times.
```

It is an error if there is no change for the cursor to move to.

4. Diff copying copy-diffs E99 E100 E101 E102 E103

merge

There are two commands to copy text from one buffer to another. The result is that the buffers will be equal within the specified range.

:diffg :diffget

```
:[range]diffg[et] [bufspec]
                Modify the current buffer to undo difference with another
                buffer. If [bufspec] is given, that buffer is used. If
                [bufspec] refers to the current buffer then nothing happens.
                Otherwise this only works if there is one other buffer in diff
                mode.
                See below for [range].
```

:diffpu :diffput E793

```
:[range]diffpu[t] [bufspec]
                Modify another buffer to undo difference with the current
                buffer. Just like ":diffget" but the other buffer is modified
                instead of the current one.
```

When [bufspec] is omitted and there is more than one other
buffer in diff mode where 'modifiable' is set this fails.
See below for [range].

do

[count]do Same as ":diffget" without range. The "o" stands for "obtain"
("dg" can't be used, it could be the start of "dgg"!). **Note:**
this doesn't work in Visual mode.
If you give a [count], it is used as the [bufspec] argument
for ":diffget".

dp

[count]dp Same as ":diffput" without range. **Note:** this doesn't work in
Visual mode.
If you give a [count], it is used as the [bufspec] argument
for ":diffput".

When no [range] is given, the diff at the cursor position or just above it is affected. There can be deleted lines below the last line of the buffer. When the cursor is on the last line in the buffer and there is no diff above this line, and no [range] is given, the diff below the cursor position will be used instead.

When [range] is used, Vim tries to only put or get the specified lines. When there are deleted lines, they will be used if they are between the lines specified by [range].

To be able to put or get those lines to/from another buffer in a [range] it's allowed to use 0 and the last line number plus one. This command gets all diffs from the other buffer:

```
:0,$+1diffget
```

Note that deleted lines are displayed, but not counted as text lines. You can't move the cursor into them. To fill the deleted lines with the lines from another buffer use ":diffget" on the line below them.

E787

When the buffer that is about to be modified is read-only and the autocommand that is triggered by FileChangedRO changes buffers the command will fail. The autocommand must not change buffers.

The [bufspec] argument above can be a buffer number, a pattern for a buffer name or a part of a buffer name. Examples:

```
:diffget               Use the other buffer which is in diff mode
:diffget 3             Use buffer 3
:diffget v2            Use the buffer which matches "v2" and is in
                       diff mode (e.g., "file.c.v2")
```

5. Diff anchors

diff-anchors E1549

Diff anchors allow you to control where the diff algorithm aligns and synchronize text across files. Each anchor matches each other in each file, allowing you to control the output of a diff.

This is useful when a change involves complicated edits. For example, if a function was moved to another location and further edited. By default, the algorithm aims to create the smallest diff, which results in that entire function being considered to be deleted and added on the other side, making it hard to see what the actual edit on it was. You can use diff anchors to pin that function so the diff algorithm will align based on it.

To use it, set anchors using 'diffanchors' which is a comma-separated list of {address} in each file, and then add "anchor" to 'diffopt'. Internally, Vim splits each file up into sections split by the anchors. It performs the diff on each pair of sections separately before merging the results back.

Setting 'diffanchors' will update the diff immediately. If an anchor is tied to a mark, and you change what the mark is pointed to, you need to manually call :diffupdate afterwards to get the updated diff results.

Example:

Let's say we have the following files, side-by-side. We are interested in the change that happened to the function foo(), which was both edited and moved.

File A:

```
int foo() {
    int n = 1;
    return n;
}

int g = 1;

int bar(int a) {
    a *= 2;
    a += 3;
    return a;
}
```

<File B:

```
int bar(int a) {
    a *= 2;
    a += 3;
    return a;
}

int foo() {
    int n = 999;
    return n;
}

int g = 1;
```

A normal diff will usually align the diff result as such:

```
int foo() {     |----------------
    int n = 1;  |----------------
    return n;   |----------------
}               |----------------
                |----------------
int g = 1;      |----------------
                |----------------
int bar(int a) {|int bar(int a) {
    a *= 2;     |    a *= 2;
    a += 3;     |    a += 3;
    return a;   |    return a;
}               |}
----------------|
----------------|int foo() {
----------------|    int n = 999;
----------------|    return n;
----------------|}
----------------|
----------------|int g = 1;
```

What we want is to instead ask the diff to align on foo():

```
----------------|int bar(int a) {
----------------|    a *= 2;
----------------|    a += 3;
----------------|    return a;
----------------|}
----------------|
int foo() {     |int foo() {
    int n = 1;  |    int n = 999;
    return n;   |    return n;
}               |}
                |
int g = 1;      |int g = 1;
                |----------------
int bar(int a) {|----------------
    a *= 2;     |----------------
    a += 3;     |----------------
    return a;   |----------------
}               |----------------
```

Below are some ways of setting diff anchors to get the above result. In each example, 'diffopt' needs to have anchor set for this to take effect.

Marks: Set the 'a mark on the int foo() lines in each file first before setting the anchors:

```
set diffanchors='a
```

Pattern: Specify the anchor using a pattern (see :/). Here, we make sure to always start search from line 1 for consistency:

```
set diffanchors=1/int\ foo(/
```

Selection: Use visual mode to select the entire foo() function body in each file. Here, we use two anchors. This does a better job of making sure only the function bodies are anchored against each other but not the lines after it. Note the '>+1 below. The "+1" is necessary as we want the split to happen below the last line of the function, not above:

```
set diffanchors='<,'>+1
```

Manually set two anchors using line numbers via buffer-local options:

```
setlocal diffanchors=1,5
wincmd w
setlocal diffanchors=7,11
```

6. Diff options

diff-options

Also see 'diffopt' and the "diff" item of 'fillchars'.

diff-slow diff_translations

For very long lines, the diff syntax highlighting might be slow, especially

since it tries to match all different kind of localisations. To disable
localisations and speed up the syntax highlighting, set the global variable
g:diff_translations to zero:

```
let g:diff_translations = 0
```

After setting this variable, reload the syntax script:

```
set syntax=diff
```

FINDING THE DIFFERENCES

diff-diffexpr

The 'diffexpr' option can be set to use something else than the internal diff
support or the standard "diff" program to compare two files and find the
differences.

When 'diffexpr' is empty, Vim uses this command to find the differences
between file1 and file2:

```
diff file1 file2 > outfile
```

The ">" is replaced with the value of 'shellredir'.

The output of "diff" must be a normal "ed" style diff or a unified diff. A
context diff will NOT work. For a unified diff no context lines can be used.
Using "diff -u" will NOT work, use "diff -U0".

This example explains the format that Vim expects for the "ed" style diff:

```
1a2
> bbb
4d4
< 111
7c7
< GGG
---
> ggg
```

The "1a2" item appends the line "bbb".
The "4d4" item deletes the line "111".
The "7c7" item replaces the line "GGG" with "ggg".

When 'diffexpr' is not empty, Vim evaluates it to obtain a diff file in the
format mentioned. These variables are set to the file names used:

```
        v:fname_in              original file
        v:fname_new             new version of the same file
        v:fname_out             where to write the resulting diff file
```

Additionally, 'diffexpr' should take care of "icase" and "iwhite" in the
'diffopt' option. 'diffexpr' cannot change the value of 'lines' and
'columns'.

The advantage of using a function call without arguments is that it is faster,
see expr-option-function.

Example (this does almost the same as 'diffexpr' being empty):

```
vim.o.diffexpr = function()
  local opt = ''
  if vim.o.diffopt:find('icase') then
    opt = opt .. '-i '
  end
  if vim.o.diffopt:find('iwhite') then
    opt = opt .. '-b '
  end
  vim.fn.system(('diff -a %s%s %s > %s'):format(
    opt, vim.v.fname_in, vim.v.fname_new, vim.v.fname_out))
end
```

The "-a" argument is used to force comparing the files as text, comparing as
binaries isn't useful.

If the 'diffexpr' expression starts with s: or <SID>, then it is replaced
with the script ID (local-function). Example:

```
set diffexpr=s:MyDiffExpr()
set diffexpr=<SID>SomeDiffExpr()
```

Otherwise, the expression is evaluated in the context of the script where the
option was set, thus script-local items are available.

E810 E97

Vim will do a test if the diff output looks alright. If it doesn't, you will
get an error message. Possible causes:

- The "diff" program cannot be executed.

- The "diff" program doesn't produce normal "ed" style diffs (see above).

- The 'shell' and associated options are not set correctly. Try if filtering
  works with a command like ":!sort".
- You are using 'diffexpr' and it doesn't work.

If it's not clear what the problem is set the 'verbose' option to one or more to see more messages.

The self-installing Vim for MS-Windows includes a diff program. If you don't have it you might want to download a diff.exe. For example from https://gnuwin32.sourceforge.net/packages/diffutils.htm.

USING PATCHES diff-patchexpr

The 'patchexpr' option can be set to use something else than the standard "patch" program.

When 'patchexpr' is empty, Vim will call the "patch" program like this:

```
patch -o outfile origfile < patchfile
```

This should work fine with most versions of the "patch" program. Note that a CR in the middle of a line may cause problems, it is seen as a line break.

If the default doesn't work for you, set the 'patchexpr' to an expression that will have the same effect. These variables are set to the file names used:

v:fname_in	original file
v:fname_diff	patch file
v:fname_out	resulting patched file

The advantage of using a function call without arguments is that it is faster, see expr-option-function.

Example (this does the same as 'patchexpr' being empty):

```
vim.o.patchexpr = function()
  vim.fn.system(('patch -o %s %s < %s'):format(
    vim.v.fname_out, vim.v.fname_in, vim.v.fname_diff))
end
```

Make sure that using the "patch" program doesn't have unwanted side effects. For example, watch out for additionally generated files, which should be deleted. It should just patch the file and nothing else.
   Vim will change directory to "/tmp" or another temp directory before evaluating 'patchexpr'. This hopefully avoids that files in the current directory are accidentally patched. Vim will also delete files starting with v:fname_in and ending in ".rej" and ".orig".

If the 'patchexpr' expression starts with s: or <SID>, then it is replaced with the script ID (local-function). Example:

```
set patchexpr=s:MyPatchExpr()
set patchexpr=<SID>SomePatchExpr()
```

Otherwise, the expression is evaluated in the context of the script where the option was set, thus script-local items are available.

Digraphs digraphs Digraphs

Digraphs are used to enter characters that normally cannot be entered by
an ordinary keyboard. These are mostly printable non-ASCII characters. The
digraphs are easier to remember than the decimal number that can be entered
with CTRL-V (see i_CTRL-V).

There is a brief introduction on digraphs in the user manual: 24.9
An alternative is using the 'keymap' option.

1. Defining digraphs digraphs-define

:dig :digraphs

:dig[raphs][!] Show currently defined digraphs.
With [!] headers are used to make it a bit easier to
find a specific character.

E104 E39

:dig[raphs] {char1}{char2} {number} ...
Add digraph {char1}{char2} to the list. {number} is
the decimal representation of the character. Normally
it is the Unicode character, see digraph-encoding.
Example:

```
:digr e: 235 a: 228
```

You can use :exe to enter a hex number:

```
:exe 'digr += ' .. 0x2A72
```

Avoid defining a digraph with '_' (underscore) as the
first character, it has a special meaning in the
future.
NOTE: This command cannot add a digraph that starts
with a white space. If you want to add such digraph,
you can use digraph_set() instead.

Example of the output of ":digraphs":

```
TH Þ  222  ss ß  223  a! à  224  a' á  225  a> â  226  a? ã  227  a: ä  228
```

The first two characters in each column are the characters you have to type to
enter the digraph.

In the middle of each column is the resulting character. This may be mangled
if you look at it on a system that does not support digraphs or if you print
this file.

digraph-encoding

The decimal number normally is the Unicode number of the character. Note that
the meaning doesn't change when 'encoding' changes. The character will be
converted from Unicode to 'encoding' when needed. This does require the
conversion to be available, it might fail. For the NUL character you will see
"10". That's because NUL characters are internally represented with a NL
character. When you write the file it will become a NUL character.

Example:

```
digraph oe 339
```

This defines the "oe" digraph for a character that is number 339 in Unicode.

2. Using digraphs digraphs-use

There are two methods to enter digraphs: i_digraph

```
CTRL-K {char1} {char2}          or
{char1} <BS> {char2}
```

The first is always available; the second only when the 'digraph' option is
set.

If a digraph with {char1}{char2} does not exist, Vim searches for a digraph
{char2}{char1}. This helps when you don't remember which character comes
first.

Note that when you enter CTRL-K {char1}, where {char1} is a special key, Vim
enters the code for that special key. This is not a digraph.

Once you have entered the digraph, Vim treats the character like a normal
character that occupies only one character in the file and on the screen.
Example:

```
'B' <BS> 'B'    will enter the broken '|' character (166)
'a' <BS> '>'    will enter an 'a' with a circumflex (226)
CTRL-K '-' '-'  will enter a soft hyphen (173)
```

The current digraphs are listed with the ":digraphs" command. Some of the
default ones are listed below digraph-table.

For CTRL-K, there is one general digraph: CTRL-K <Space> {char} will enter
{char} with the highest bit set. You can use this to enter meta-characters.

The <Esc> character cannot be part of a digraph. When hitting <Esc>, Vim stops digraph entry and ends Insert mode or Command-line mode, just like hitting an <Esc> out of digraph context. Use CTRL-V 155 to enter meta-ESC

(CSI).

If you accidentally typed an 'a' that should be an 'e', you will type 'a' <BS> 'e'. But that is a digraph, so you will not get what you want. To correct this, you will have to type <BS> e again. To avoid this don't set the 'digraph' option and use CTRL-K to enter digraphs.

You may have problems using Vim with characters which have a value above 128. For example: You insert ue (u-umlaut) and the editor echoes \334 in Insert mode. After leaving the Insert mode everything is fine. On some Unix systems this means you have to define the environment-variable LC_CTYPE. If you are using csh, then put the following line in your .cshrc:

```
setenv LC_CTYPE en_US.utf8
```

(or similar for a different language or country). The value must be a valid locale on your system, i.e. on Unix-like systems it must be present in the output of

```
locale -a
```

3. Default digraphs

digraphs-default

Vim comes with a set of default digraphs. Check the output of ":digraphs" to see them.

On most systems Vim uses the same digraphs. They work for the Unicode and ISO-8859-1 character sets. These default digraphs are taken from the RFC1345 mnemonics (with some additions). To make it easy to remember the mnemonic, the second character has a standard meaning:

char name	char	meaning
Exclamation mark	!	Grave
Apostrophe	'	Acute accent
Greater-Than sign	>	Circumflex accent
Question mark	?	Tilde
Hyphen-Minus	-	Macron
Left parenthesis	(	Breve
Full stop	.	Dot above
Colon	:	Diaeresis
Comma	,	Cedilla
Underline	_	Underline
Solidus	/	Stroke
Quotation mark	"	Double acute accent
Semicolon	;	Ogonek
Less-Than sign	<	Caron
Zero	0	Ring above
Two	2	Hook
Nine	9	Horn

```
       Equals                =       Cyrillic (= used as second char)
       Asterisk              *       Greek
       Percent sign          %       Greek/Cyrillic special
       Plus                  +       smalls: Arabic, capitals: Hebrew
       Three                 3       some Latin/Greek/Cyrillic letters
       Four                  4       Bopomofo
       Five                  5       Hiragana
       Six                   6       Katakana
```

Example: a: is ä and o: is ö

These are the RFC1345 digraphs for the one-byte characters. See the output of ":digraphs" for the others.

EURO

euro euro-digraph

Exception: RFC1345 doesn't specify the euro sign. In Vim the digraph =e was added for this. Note the difference between latin1, where the digraph Cu is used for the currency sign, and latin9 (iso-8859-15), where the digraph =e is used for the euro sign, while both of them are the character 164, 0xa4. For compatibility with zsh Eu can also be used for the euro sign.

ROUBLE

The rouble sign was added in 2014 as 0x20bd. Vim supports the digraphs =R and =P for this. Note that R= and P= are other characters.

QUADRUPLE PRIME

The quadruple prime using the digraph 4' was added in 2023. Although it is not part of RFC 1345, it supplements the existing digraph implementation as there already exist digraphs for PRIME, DOUBLE PRIME and TRIPLE PRIME using the 1', 2' and 3' digraphs.

digraph-table digraph-table-mbyte

char	digraph	hex	dec	official name
^@	NU	0x00	0	NULL (NUL)
^A	SH	0x01	1	START OF HEADING (SOH)
^B	SX	0x02	2	START OF TEXT (STX)
^C	EX	0x03	3	END OF TEXT (ETX)
^D	ET	0x04	4	END OF TRANSMISSION (EOT)
^E	EQ	0x05	5	ENQUIRY (ENQ)
^F	AK	0x06	6	ACKNOWLEDGE (ACK)
^G	BL	0x07	7	BELL (BEL)
^H	BS	0x08	8	BACKSPACE (BS)
^I	HT	0x09	9	CHARACTER TABULATION (HT)
^@	LF	0x0a	10	LINE FEED (LF)
^K	VT	0x0b	11	LINE TABULATION (VT)
^L	FF	0x0c	12	FORM FEED (FF)
^M	CR	0x0d	13	CARRIAGE RETURN (CR)
^N	SO	0x0e	14	SHIFT OUT (SO)
^O	SI	0x0f	15	SHIFT IN (SI)
^P	DL	0x10	16	DATALINK ESCAPE (DLE)
^Q	D1	0x11	17	DEVICE CONTROL ONE (DC1)
^R	D2	0x12	18	DEVICE CONTROL TWO (DC2)
^S	D3	0x13	19	DEVICE CONTROL THREE (DC3)
^T	D4	0x14	20	DEVICE CONTROL FOUR (DC4)
^U	NK	0x15	21	NEGATIVE ACKNOWLEDGE (NAK)
^V	SY	0x16	22	SYNCHRONOUS IDLE (SYN)
^W	EB	0x17	23	END OF TRANSMISSION BLOCK (ETB)
^X	CN	0x18	24	CANCEL (CAN)
^Y	EM	0x19	25	END OF MEDIUM (EM)
^Z	SB	0x1a	26	SUBSTITUTE (SUB)
^[	EC	0x1b	27	ESCAPE (ESC)
^\	FS	0x1c	28	FILE SEPARATOR (IS4)
^]	GS	0x1d	29	GROUP SEPARATOR (IS3)
^^	RS	0x1e	30	RECORD SEPARATOR (IS2)
^_	US	0x1f	31	UNIT SEPARATOR (IS1)
	SP	0x20	32	SPACE
#	Nb	0x23	35	NUMBER SIGN
$	DO	0x24	36	DOLLAR SIGN
@	At	0x40	64	COMMERCIAL AT
[	<(	0x5b	91	LEFT SQUARE BRACKET
\	//	0x5c	92	REVERSE SOLIDUS
]	)>	0x5d	93	RIGHT SQUARE BRACKET
^	'>	0x5e	94	CIRCUMFLEX ACCENT
`	'!	0x60	96	GRAVE ACCENT
{	(!	0x7b	123	LEFT CURLY BRACKET
|	!!	0x7c	124	VERTICAL LINE
}	!)	0x7d	125	RIGHT CURLY BRACKET
~	'?	0x7e	126	TILDE
^?	DT	0x7f	127	DELETE (DEL)
~@	PA	0x80	128	PADDING CHARACTER (PAD)
~A	HO	0x81	129	HIGH OCTET PRESET (HOP)
~B	BH	0x82	130	BREAK PERMITTED HERE (BPH)
~C	NH	0x83	131	NO BREAK HERE (NBH)
~D	IN	0x84	132	INDEX (IND)
~E	NL	0x85	133	NEXT LINE (NEL)
~F	SA	0x86	134	START OF SELECTED AREA (SSA)
~G	ES	0x87	135	END OF SELECTED AREA (ESA)
~H	HS	0x88	136	CHARACTER TABULATION SET (HTS)
~I	HJ	0x89	137	CHARACTER TABULATION WITH JUSTIFICATION (HTJ)
~J	VS	0x8a	138	LINE TABULATION SET (VTS)
~K	PD	0x8b	139	PARTIAL LINE FORWARD (PLD)
~L	PU	0x8c	140	PARTIAL LINE BACKWARD (PLU)
~M	RI	0x8d	141	REVERSE LINE FEED (RI)
~N	S2	0x8e	142	SINGLE-SHIFT TWO (SS2)
~O	S3	0x8f	143	SINGLE-SHIFT THREE (SS3)
~P	DC	0x90	144	DEVICE CONTROL STRING (DCS)
~Q	P1	0x91	145	PRIVATE USE ONE (PU1)
~R	P2	0x92	146	PRIVATE USE TWO (PU2)
~S	TS	0x93	147	SET TRANSMIT STATE (STS)
~T	CC	0x94	148	CANCEL CHARACTER (CCH)
~U	MW	0x95	149	MESSAGE WAITING (MW)
~V	SG	0x96	150	START OF GUARDED AREA (SPA)
~W	EG	0x97	151	END OF GUARDED AREA (EPA)
~X	SS	0x98	152	START OF STRING (SOS)
~Y	GC	0x99	153	SINGLE GRAPHIC CHARACTER INTRODUCER (SGCI)
~Z	SC	0x9a	154	SINGLE CHARACTER INTRODUCER (SCI)
~[	CI	0x9b	155	CONTROL SEQUENCE INTRODUCER (CSI)
~\	ST	0x9c	156	STRING TERMINATOR (ST)
~]	OC	0x9d	157	OPERATING SYSTEM COMMAND (OSC)
~^	PM	0x9e	158	PRIVACY MESSAGE (PM)
~_	AC	0x9f	159	APPLICATION PROGRAM COMMAND (APC)
|	NS	0xa0	160	NO-BREAK SPACE
¡	!I	0xa1	161	INVERTED EXCLAMATION MARK
¢	Ct	0xa2	162	CENT SIGN
£	Pd	0xa3	163	POUND SIGN
¤	Cu	0xa4	164	CURRENCY SIGN
¥	Ye	0xa5	165	YEN SIGN
¦	BB	0xa6	166	BROKEN BAR
§	SE	0xa7	167	SECTION SIGN
¨	':	0xa8	168	DIAERESIS
©	Co	0xa9	169	COPYRIGHT SIGN

ª	-a	0xaa	170	FEMININE ORDINAL INDICATOR
«	<<	0xab	171	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
¬	NO	0xac	172	NOT SIGN
	--	0xad	173	SOFT HYPHEN
®	Rg	0xae	174	REGISTERED SIGN
¯	'm	0xaf	175	MACRON
°	DG	0xb0	176	DEGREE SIGN
±	+-	0xb1	177	PLUS-MINUS SIGN
²	2S	0xb2	178	SUPERSCRIPT TWO
³	3S	0xb3	179	SUPERSCRIPT THREE
´	''	0xb4	180	ACUTE ACCENT
µ	My	0xb5	181	MICRO SIGN
¶	PI	0xb6	182	PILCROW SIGN
·	.M	0xb7	183	MIDDLE DOT
¸	',	0xb8	184	CEDILLA
¹	1S	0xb9	185	SUPERSCRIPT ONE
º	-o	0xba	186	MASCULINE ORDINAL INDICATOR
»	>>	0xbb	187	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
¼	14	0xbc	188	VULGAR FRACTION ONE QUARTER
½	12	0xbd	189	VULGAR FRACTION ONE HALF
¾	34	0xbe	190	VULGAR FRACTION THREE QUARTERS
¿	?I	0xbf	191	INVERTED QUESTION MARK
À	A!	0xc0	192	LATIN CAPITAL LETTER A WITH GRAVE
Á	A'	0xc1	193	LATIN CAPITAL LETTER A WITH ACUTE
Â	A>	0xc2	194	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
Ã	A?	0xc3	195	LATIN CAPITAL LETTER A WITH TILDE
Ä	A:	0xc4	196	LATIN CAPITAL LETTER A WITH DIAERESIS
Å	AA	0xc5	197	LATIN CAPITAL LETTER A WITH RING ABOVE
Æ	AE	0xc6	198	LATIN CAPITAL LETTER AE
Ç	C,	0xc7	199	LATIN CAPITAL LETTER C WITH CEDILLA
È	E!	0xc8	200	LATIN CAPITAL LETTER E WITH GRAVE
É	E'	0xc9	201	LATIN CAPITAL LETTER E WITH ACUTE
Ê	E>	0xca	202	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
Ë	E:	0xcb	203	LATIN CAPITAL LETTER E WITH DIAERESIS
Ì	I!	0xcc	204	LATIN CAPITAL LETTER I WITH GRAVE
Í	I'	0xcd	205	LATIN CAPITAL LETTER I WITH ACUTE
Î	I>	0xce	206	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
Ï	I:	0xcf	207	LATIN CAPITAL LETTER I WITH DIAERESIS
Ð	D-	0xd0	208	LATIN CAPITAL LETTER ETH (Icelandic)
Ñ	N?	0xd1	209	LATIN CAPITAL LETTER N WITH TILDE
Ò	O!	0xd2	210	LATIN CAPITAL LETTER O WITH GRAVE
Ó	O'	0xd3	211	LATIN CAPITAL LETTER O WITH ACUTE
Ô	O>	0xd4	212	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
Õ	O?	0xd5	213	LATIN CAPITAL LETTER O WITH TILDE
Ö	O:	0xd6	214	LATIN CAPITAL LETTER O WITH DIAERESIS
×	*X	0xd7	215	MULTIPLICATION SIGN
Ø	O/	0xd8	216	LATIN CAPITAL LETTER O WITH STROKE
Ù	U!	0xd9	217	LATIN CAPITAL LETTER U WITH GRAVE
Ú	U'	0xda	218	LATIN CAPITAL LETTER U WITH ACUTE
Û	U>	0xdb	219	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
Ü	U:	0xdc	220	LATIN CAPITAL LETTER U WITH DIAERESIS
Ý	Y'	0xdd	221	LATIN CAPITAL LETTER Y WITH ACUTE
Þ	TH	0xde	222	LATIN CAPITAL LETTER THORN (Icelandic)
ß	ss	0xdf	223	LATIN SMALL LETTER SHARP S (German)
à	a!	0xe0	224	LATIN SMALL LETTER A WITH GRAVE
á	a'	0xe1	225	LATIN SMALL LETTER A WITH ACUTE
â	a>	0xe2	226	LATIN SMALL LETTER A WITH CIRCUMFLEX
ã	a?	0xe3	227	LATIN SMALL LETTER A WITH TILDE
ä	a:	0xe4	228	LATIN SMALL LETTER A WITH DIAERESIS
å	aa	0xe5	229	LATIN SMALL LETTER A WITH RING ABOVE
æ	ae	0xe6	230	LATIN SMALL LETTER AE
ç	c,	0xe7	231	LATIN SMALL LETTER C WITH CEDILLA
è	e!	0xe8	232	LATIN SMALL LETTER E WITH GRAVE
é	e'	0xe9	233	LATIN SMALL LETTER E WITH ACUTE
ê	e>	0xea	234	LATIN SMALL LETTER E WITH CIRCUMFLEX
ë	e:	0xeb	235	LATIN SMALL LETTER E WITH DIAERESIS
ì	i!	0xec	236	LATIN SMALL LETTER I WITH GRAVE
í	i'	0xed	237	LATIN SMALL LETTER I WITH ACUTE
î	i>	0xee	238	LATIN SMALL LETTER I WITH CIRCUMFLEX
ï	i:	0xef	239	LATIN SMALL LETTER I WITH DIAERESIS
ð	d-	0xf0	240	LATIN SMALL LETTER ETH (Icelandic)
ñ	n?	0xf1	241	LATIN SMALL LETTER N WITH TILDE
ò	o!	0xf2	242	LATIN SMALL LETTER O WITH GRAVE
ó	o'	0xf3	243	LATIN SMALL LETTER O WITH ACUTE
ô	o>	0xf4	244	LATIN SMALL LETTER O WITH CIRCUMFLEX
õ	o?	0xf5	245	LATIN SMALL LETTER O WITH TILDE
ö	o:	0xf6	246	LATIN SMALL LETTER O WITH DIAERESIS
÷	-:	0xf7	247	DIVISION SIGN
ø	o/	0xf8	248	LATIN SMALL LETTER O WITH STROKE
ù	u!	0xf9	249	LATIN SMALL LETTER U WITH GRAVE
ú	u'	0xfa	250	LATIN SMALL LETTER U WITH ACUTE
û	u>	0xfb	251	LATIN SMALL LETTER U WITH CIRCUMFLEX
ü	u:	0xfc	252	LATIN SMALL LETTER U WITH DIAERESIS
ý	y'	0xfd	253	LATIN SMALL LETTER Y WITH ACUTE
þ	th	0xfe	254	LATIN SMALL LETTER THORN (Icelandic)
ÿ	y:	0xff	255	LATIN SMALL LETTER Y WITH DIAERESIS
Ā	A-	0100	0256	LATIN CAPITAL LETTER A WITH MACRON
ā	a-	0101	0257	LATIN SMALL LETTER A WITH MACRON
Ă	A(	0102	0258	LATIN CAPITAL LETTER A WITH BREVE
ă	a(	0103	0259	LATIN SMALL LETTER A WITH BREVE

Ą	A;	0104	0260	LATIN CAPITAL LETTER A WITH OGONEK
ą	a;	0105	0261	LATIN SMALL LETTER A WITH OGONEK
Ć	C'	0106	0262	LATIN CAPITAL LETTER C WITH ACUTE
ć	c'	0107	0263	LATIN SMALL LETTER C WITH ACUTE
Ĉ	C>	0108	0264	LATIN CAPITAL LETTER C WITH CIRCUMFLEX
ĉ	c>	0109	0265	LATIN SMALL LETTER C WITH CIRCUMFLEX
Ċ	C.	010A	0266	LATIN CAPITAL LETTER C WITH DOT ABOVE
ċ	c.	010B	0267	LATIN SMALL LETTER C WITH DOT ABOVE
Č	C<	010C	0268	LATIN CAPITAL LETTER C WITH CARON
č	c<	010D	0269	LATIN SMALL LETTER C WITH CARON
Ď	D<	010E	0270	LATIN CAPITAL LETTER D WITH CARON
ď	d<	010F	0271	LATIN SMALL LETTER D WITH CARON
Đ	D/	0110	0272	LATIN CAPITAL LETTER D WITH STROKE
đ	d/	0111	0273	LATIN SMALL LETTER D WITH STROKE
Ē	E-	0112	0274	LATIN CAPITAL LETTER E WITH MACRON
ē	e-	0113	0275	LATIN SMALL LETTER E WITH MACRON
Ĕ	E(	0114	0276	LATIN CAPITAL LETTER E WITH BREVE
ĕ	e(	0115	0277	LATIN SMALL LETTER E WITH BREVE
Ė	E.	0116	0278	LATIN CAPITAL LETTER E WITH DOT ABOVE
ė	e.	0117	0279	LATIN SMALL LETTER E WITH DOT ABOVE
Ę	E;	0118	0280	LATIN CAPITAL LETTER E WITH OGONEK
ę	e;	0119	0281	LATIN SMALL LETTER E WITH OGONEK
Ě	E<	011A	0282	LATIN CAPITAL LETTER E WITH CARON
ě	e<	011B	0283	LATIN SMALL LETTER E WITH CARON
Ĝ	G>	011C	0284	LATIN CAPITAL LETTER G WITH CIRCUMFLEX
ĝ	g>	011D	0285	LATIN SMALL LETTER G WITH CIRCUMFLEX
Ğ	G(	011E	0286	LATIN CAPITAL LETTER G WITH BREVE
ğ	g(	011F	0287	LATIN SMALL LETTER G WITH BREVE
Ġ	G.	0120	0288	LATIN CAPITAL LETTER G WITH DOT ABOVE
ġ	g.	0121	0289	LATIN SMALL LETTER G WITH DOT ABOVE
Ģ	G,	0122	0290	LATIN CAPITAL LETTER G WITH CEDILLA
ģ	g,	0123	0291	LATIN SMALL LETTER G WITH CEDILLA
Ĥ	H>	0124	0292	LATIN CAPITAL LETTER H WITH CIRCUMFLEX
ĥ	h>	0125	0293	LATIN SMALL LETTER H WITH CIRCUMFLEX
Ħ	H/	0126	0294	LATIN CAPITAL LETTER H WITH STROKE
ħ	h/	0127	0295	LATIN SMALL LETTER H WITH STROKE
Ĩ	I?	0128	0296	LATIN CAPITAL LETTER I WITH TILDE
ĩ	i?	0129	0297	LATIN SMALL LETTER I WITH TILDE
Ī	I-	012A	0298	LATIN CAPITAL LETTER I WITH MACRON
ī	i-	012B	0299	LATIN SMALL LETTER I WITH MACRON
Ĭ	I(	012C	0300	LATIN CAPITAL LETTER I WITH BREVE
ĭ	i(	012D	0301	LATIN SMALL LETTER I WITH BREVE
Į	I;	012E	0302	LATIN CAPITAL LETTER I WITH OGONEK
į	i;	012F	0303	LATIN SMALL LETTER I WITH OGONEK
İ	I.	0130	0304	LATIN CAPITAL LETTER I WITH DOT ABOVE
ı	i.	0131	0305	LATIN SMALL LETTER DOTLESS I
Ĳ	IJ	0132	0306	LATIN CAPITAL LIGATURE IJ
ĳ	ij	0133	0307	LATIN SMALL LIGATURE IJ
Ĵ	J>	0134	0308	LATIN CAPITAL LETTER J WITH CIRCUMFLEX
ĵ	j>	0135	0309	LATIN SMALL LETTER J WITH CIRCUMFLEX
Ķ	K,	0136	0310	LATIN CAPITAL LETTER K WITH CEDILLA
ķ	k,	0137	0311	LATIN SMALL LETTER K WITH CEDILLA
ĸ	kk	0138	0312	LATIN SMALL LETTER KRA
Ĺ	L'	0139	0313	LATIN CAPITAL LETTER L WITH ACUTE
ĺ	l'	013A	0314	LATIN SMALL LETTER L WITH ACUTE
Ļ	L,	013B	0315	LATIN CAPITAL LETTER L WITH CEDILLA
ļ	l,	013C	0316	LATIN SMALL LETTER L WITH CEDILLA
Ľ	L<	013D	0317	LATIN CAPITAL LETTER L WITH CARON
ľ	l<	013E	0318	LATIN SMALL LETTER L WITH CARON
Ŀ	L.	013F	0319	LATIN CAPITAL LETTER L WITH MIDDLE DOT
ŀ	l.	0140	0320	LATIN SMALL LETTER L WITH MIDDLE DOT
Ł	L/	0141	0321	LATIN CAPITAL LETTER L WITH STROKE
ł	l/	0142	0322	LATIN SMALL LETTER L WITH STROKE
Ń	N'	0143	0323	LATIN CAPITAL LETTER N WITH ACUTE `
ń	n'	0144	0324	LATIN SMALL LETTER N WITH ACUTE `
Ņ	N,	0145	0325	LATIN CAPITAL LETTER N WITH CEDILLA `
ņ	n,	0146	0326	LATIN SMALL LETTER N WITH CEDILLA `
Ň	N<	0147	0327	LATIN CAPITAL LETTER N WITH CARON `
ň	n<	0148	0328	LATIN SMALL LETTER N WITH CARON `
ŉ	'n	0149	0329	LATIN SMALL LETTER N PRECEDED BY APOSTROPHE `
Ŋ	NG	014A	0330	LATIN CAPITAL LETTER ENG
ŋ	ng	014B	0331	LATIN SMALL LETTER ENG
Ō	O-	014C	0332	LATIN CAPITAL LETTER O WITH MACRON
ō	o-	014D	0333	LATIN SMALL LETTER O WITH MACRON
Ŏ	O(	014E	0334	LATIN CAPITAL LETTER O WITH BREVE
ŏ	o(	014F	0335	LATIN SMALL LETTER O WITH BREVE
Ő	O"	0150	0336	LATIN CAPITAL LETTER O WITH DOUBLE ACUTE
ő	o"	0151	0337	LATIN SMALL LETTER O WITH DOUBLE ACUTE
Œ	OE	0152	0338	LATIN CAPITAL LIGATURE OE
œ	oe	0153	0339	LATIN SMALL LIGATURE OE
Ŕ	R'	0154	0340	LATIN CAPITAL LETTER R WITH ACUTE
ŕ	r'	0155	0341	LATIN SMALL LETTER R WITH ACUTE
Ŗ	R,	0156	0342	LATIN CAPITAL LETTER R WITH CEDILLA
ŗ	r,	0157	0343	LATIN SMALL LETTER R WITH CEDILLA
Ř	R<	0158	0344	LATIN CAPITAL LETTER R WITH CARON
ř	r<	0159	0345	LATIN SMALL LETTER R WITH CARON
Ś	S'	015A	0346	LATIN CAPITAL LETTER S WITH ACUTE
ś	s'	015B	0347	LATIN SMALL LETTER S WITH ACUTE
Ŝ	S>	015C	0348	LATIN CAPITAL LETTER S WITH CIRCUMFLEX
ŝ	s>	015D	0349	LATIN SMALL LETTER S WITH CIRCUMFLEX

Ş	S,	015E	0350	LATIN CAPITAL LETTER S WITH CEDILLA
ş	s,	015F	0351	LATIN SMALL LETTER S WITH CEDILLA
Š	S<	0160	0352	LATIN CAPITAL LETTER S WITH CARON
š	s<	0161	0353	LATIN SMALL LETTER S WITH CARON
Ţ	T,	0162	0354	LATIN CAPITAL LETTER T WITH CEDILLA
ţ	t,	0163	0355	LATIN SMALL LETTER T WITH CEDILLA
Ť	T<	0164	0356	LATIN CAPITAL LETTER T WITH CARON
ť	t<	0165	0357	LATIN SMALL LETTER T WITH CARON
Ŧ	T/	0166	0358	LATIN CAPITAL LETTER T WITH STROKE
ŧ	t/	0167	0359	LATIN SMALL LETTER T WITH STROKE
Ũ	U?	0168	0360	LATIN CAPITAL LETTER U WITH TILDE
ũ	u?	0169	0361	LATIN SMALL LETTER U WITH TILDE
Ū	U-	016A	0362	LATIN CAPITAL LETTER U WITH MACRON
ū	u-	016B	0363	LATIN SMALL LETTER U WITH MACRON
Ŭ	U(	016C	0364	LATIN CAPITAL LETTER U WITH BREVE
ŭ	u(	016D	0365	LATIN SMALL LETTER U WITH BREVE
Ů	U0	016E	0366	LATIN CAPITAL LETTER U WITH RING ABOVE
ů	u0	016F	0367	LATIN SMALL LETTER U WITH RING ABOVE
Ű	U"	0170	0368	LATIN CAPITAL LETTER U WITH DOUBLE ACUTE
ű	u"	0171	0369	LATIN SMALL LETTER U WITH DOUBLE ACUTE
Ų	U;	0172	0370	LATIN CAPITAL LETTER U WITH OGONEK
ų	u;	0173	0371	LATIN SMALL LETTER U WITH OGONEK
Ŵ	W>	0174	0372	LATIN CAPITAL LETTER W WITH CIRCUMFLEX
ŵ	w>	0175	0373	LATIN SMALL LETTER W WITH CIRCUMFLEX
Ŷ	Y>	0176	0374	LATIN CAPITAL LETTER Y WITH CIRCUMFLEX
ŷ	y>	0177	0375	LATIN SMALL LETTER Y WITH CIRCUMFLEX
Ÿ	Y:	0178	0376	LATIN CAPITAL LETTER Y WITH DIAERESIS
Ź	Z'	0179	0377	LATIN CAPITAL LETTER Z WITH ACUTE
ź	z'	017A	0378	LATIN SMALL LETTER Z WITH ACUTE
Ż	Z.	017B	0379	LATIN CAPITAL LETTER Z WITH DOT ABOVE
ż	z.	017C	0380	LATIN SMALL LETTER Z WITH DOT ABOVE
Ž	Z<	017D	0381	LATIN CAPITAL LETTER Z WITH CARON
ž	z<	017E	0382	LATIN SMALL LETTER Z WITH CARON
Ơ	O9	01A0	0416	LATIN CAPITAL LETTER O WITH HORN
ơ	o9	01A1	0417	LATIN SMALL LETTER O WITH HORN
Ƣ	OI	01A2	0418	LATIN CAPITAL LETTER OI
ƣ	oi	01A3	0419	LATIN SMALL LETTER OI
Ʀ	yr	01A6	0422	LATIN LETTER YR
Ư	U9	01AF	0431	LATIN CAPITAL LETTER U WITH HORN
ư	u9	01B0	0432	LATIN SMALL LETTER U WITH HORN
Ƶ	Z/	01B5	0437	LATIN CAPITAL LETTER Z WITH STROKE
ƶ	z/	01B6	0438	LATIN SMALL LETTER Z WITH STROKE
Ʒ	ED	01B7	0439	LATIN CAPITAL LETTER EZH
Ǎ	A<	01CD	0461	LATIN CAPITAL LETTER A WITH CARON
ǎ	a<	01CE	0462	LATIN SMALL LETTER A WITH CARON
Ǐ	I<	01CF	0463	LATIN CAPITAL LETTER I WITH CARON
ǐ	i<	01D0	0464	LATIN SMALL LETTER I WITH CARON
Ǒ	O<	01D1	0465	LATIN CAPITAL LETTER O WITH CARON
ǒ	o<	01D2	0466	LATIN SMALL LETTER O WITH CARON
Ǔ	U<	01D3	0467	LATIN CAPITAL LETTER U WITH CARON
ǔ	u<	01D4	0468	LATIN SMALL LETTER U WITH CARON
Ǟ	A1	01DE	0478	LATIN CAPITAL LETTER A WITH DIAERESIS AND MACRON
ǟ	a1	01DF	0479	LATIN SMALL LETTER A WITH DIAERESIS AND MACRON
Ǡ	A7	01E0	0480	LATIN CAPITAL LETTER A WITH DOT ABOVE AND MACRON
ǡ	a7	01E1	0481	LATIN SMALL LETTER A WITH DOT ABOVE AND MACRON
Ǣ	A3	01E2	0482	LATIN CAPITAL LETTER AE WITH MACRON
ǣ	a3	01E3	0483	LATIN SMALL LETTER AE WITH MACRON
Ǥ	G/	01E4	0484	LATIN CAPITAL LETTER G WITH STROKE
ǥ	g/	01E5	0485	LATIN SMALL LETTER G WITH STROKE
Ǧ	G<	01E6	0486	LATIN CAPITAL LETTER G WITH CARON
ǧ	g<	01E7	0487	LATIN SMALL LETTER G WITH CARON
Ǩ	K<	01E8	0488	LATIN CAPITAL LETTER K WITH CARON
ǩ	k<	01E9	0489	LATIN SMALL LETTER K WITH CARON
Ǫ	O;	01EA	0490	LATIN CAPITAL LETTER O WITH OGONEK
ǫ	o;	01EB	0491	LATIN SMALL LETTER O WITH OGONEK
Ǭ	O1	01EC	0492	LATIN CAPITAL LETTER O WITH OGONEK AND MACRON
ǭ	o1	01ED	0493	LATIN SMALL LETTER O WITH OGONEK AND MACRON
Ǯ	EZ	01EE	0494	LATIN CAPITAL LETTER EZH WITH CARON
ǯ	ez	01EF	0495	LATIN SMALL LETTER EZH WITH CARON
ǰ	j<	01F0	0496	LATIN SMALL LETTER J WITH CARON
Ǵ	G'	01F4	0500	LATIN CAPITAL LETTER G WITH ACUTE
ǵ	g'	01F5	0501	LATIN SMALL LETTER G WITH ACUTE
ʿ	;S	02BF	0703	MODIFIER LETTER LEFT HALF RING
ˇ	'<	02C7	0711	CARON
˘	'(	02D8	0728	BREVE
˙	'.	02D9	0729	DOT ABOVE
˚	'0	02DA	0730	RING ABOVE
˛	';	02DB	0731	OGONEK
˝	'"	02DD	0733	DOUBLE ACUTE ACCENT
Ά	A%	0386	0902	GREEK CAPITAL LETTER ALPHA WITH TONOS
Έ	E%	0388	0904	GREEK CAPITAL LETTER EPSILON WITH TONOS
Ή	Y%	0389	0905	GREEK CAPITAL LETTER ETA WITH TONOS
Ί	I%	038A	0906	GREEK CAPITAL LETTER IOTA WITH TONOS
Ό	O%	038C	0908	GREEK CAPITAL LETTER OMICRON WITH TONOS
Ύ	U%	038E	0910	GREEK CAPITAL LETTER UPSILON WITH TONOS
Ώ	W%	038F	0911	GREEK CAPITAL LETTER OMEGA WITH TONOS
ΐ	i3	0390	0912	GREEK SMALL LETTER IOTA WITH DIALYTIKA AND TONOS
Α	A*	0391	0913	GREEK CAPITAL LETTER ALPHA
Β	B*	0392	0914	GREEK CAPITAL LETTER BETA
Γ	G*	0393	0915	GREEK CAPITAL LETTER GAMMA

Δ	D*	0394	0916	GREEK CAPITAL LETTER DELTA
Ε	E*	0395	0917	GREEK CAPITAL LETTER EPSILON
Ζ	Z*	0396	0918	GREEK CAPITAL LETTER ZETA
Η	Y*	0397	0919	GREEK CAPITAL LETTER ETA
Θ	H*	0398	0920	GREEK CAPITAL LETTER THETA
Ι	I*	0399	0921	GREEK CAPITAL LETTER IOTA
Κ	K*	039A	0922	GREEK CAPITAL LETTER KAPPA
Λ	L*	039B	0923	GREEK CAPITAL LETTER LAMDA (aka LAMBDA)
Μ	M*	039C	0924	GREEK CAPITAL LETTER MU
Ν	N*	039D	0925	GREEK CAPITAL LETTER NU
Ξ	C*	039E	0926	GREEK CAPITAL LETTER XI
Ο	O*	039F	0927	GREEK CAPITAL LETTER OMICRON
Π	P*	03A0	0928	GREEK CAPITAL LETTER PI
Ρ	R*	03A1	0929	GREEK CAPITAL LETTER RHO
Σ	S*	03A3	0931	GREEK CAPITAL LETTER SIGMA
Τ	T*	03A4	0932	GREEK CAPITAL LETTER TAU
Υ	U*	03A5	0933	GREEK CAPITAL LETTER UPSILON
Φ	F*	03A6	0934	GREEK CAPITAL LETTER PHI
Χ	X*	03A7	0935	GREEK CAPITAL LETTER CHI
Ψ	Q*	03A8	0936	GREEK CAPITAL LETTER PSI
Ω	W*	03A9	0937	GREEK CAPITAL LETTER OMEGA
Ϊ	J*	03AA	0938	GREEK CAPITAL LETTER IOTA WITH DIALYTIKA
Ϋ	V*	03AB	0939	GREEK CAPITAL LETTER UPSILON WITH DIALYTIKA
ά	a%	03AC	0940	GREEK SMALL LETTER ALPHA WITH TONOS
έ	e%	03AD	0941	GREEK SMALL LETTER EPSILON WITH TONOS
ή	y%	03AE	0942	GREEK SMALL LETTER ETA WITH TONOS
ί	i%	03AF	0943	GREEK SMALL LETTER IOTA WITH TONOS
ΰ	u3	03B0	0944	GREEK SMALL LETTER UPSILON WITH DIALYTIKA AND TONOS
α	a*	03B1	0945	GREEK SMALL LETTER ALPHA
β	b*	03B2	0946	GREEK SMALL LETTER BETA
γ	g*	03B3	0947	GREEK SMALL LETTER GAMMA
δ	d*	03B4	0948	GREEK SMALL LETTER DELTA
ε	e*	03B5	0949	GREEK SMALL LETTER EPSILON
ζ	z*	03B6	0950	GREEK SMALL LETTER ZETA
η	y*	03B7	0951	GREEK SMALL LETTER ETA
θ	h*	03B8	0952	GREEK SMALL LETTER THETA
ι	i*	03B9	0953	GREEK SMALL LETTER IOTA
κ	k*	03BA	0954	GREEK SMALL LETTER KAPPA
λ	l*	03BB	0955	GREEK SMALL LETTER LAMDA (aka LAMBDA)
μ	m*	03BC	0956	GREEK SMALL LETTER MU
ν	n*	03BD	0957	GREEK SMALL LETTER NU
ξ	c*	03BE	0958	GREEK SMALL LETTER XI
ο	o*	03BF	0959	GREEK SMALL LETTER OMICRON
π	p*	03C0	0960	GREEK SMALL LETTER PI
ρ	r*	03C1	0961	GREEK SMALL LETTER RHO
ς	*s	03C2	0962	GREEK SMALL LETTER FINAL SIGMA
σ	s*	03C3	0963	GREEK SMALL LETTER SIGMA
τ	t*	03C4	0964	GREEK SMALL LETTER TAU
υ	u*	03C5	0965	GREEK SMALL LETTER UPSILON
φ	f*	03C6	0966	GREEK SMALL LETTER PHI
χ	x*	03C7	0967	GREEK SMALL LETTER CHI
ψ	q*	03C8	0968	GREEK SMALL LETTER PSI
ω	w*	03C9	0969	GREEK SMALL LETTER OMEGA
ϊ	j*	03CA	0970	GREEK SMALL LETTER IOTA WITH DIALYTIKA
ϋ	v*	03CB	0971	GREEK SMALL LETTER UPSILON WITH DIALYTIKA
ό	o%	03CC	0972	GREEK SMALL LETTER OMICRON WITH TONOS
ύ	u%	03CD	0973	GREEK SMALL LETTER UPSILON WITH TONOS
ώ	w%	03CE	0974	GREEK SMALL LETTER OMEGA WITH TONOS
Ϙ	'G	03D8	0984	GREEK LETTER ARCHAIC KOPPA
ϙ	,G	03D9	0985	GREEK SMALL LETTER ARCHAIC KOPPA
Ϛ	T3	03DA	0986	GREEK LETTER STIGMA
ϛ	t3	03DB	0987	GREEK SMALL LETTER STIGMA
Ϝ	M3	03DC	0988	GREEK LETTER DIGAMMA
ϝ	m3	03DD	0989	GREEK SMALL LETTER DIGAMMA
Ϟ	K3	03DE	0990	GREEK LETTER KOPPA
ϟ	k3	03DF	0991	GREEK SMALL LETTER KOPPA
Ϡ	P3	03E0	0992	GREEK LETTER SAMPI
ϡ	p3	03E1	0993	GREEK SMALL LETTER SAMPI
ϴ	'%	03F4	1012	GREEK CAPITAL THETA SYMBOL
ϵ	j3	03F5	1013	GREEK LUNATE EPSILON SYMBOL
Ё	IO	0401	1025	CYRILLIC CAPITAL LETTER IO
Ђ	D%	0402	1026	CYRILLIC CAPITAL LETTER DJE
Ѓ	G%	0403	1027	CYRILLIC CAPITAL LETTER GJE
Є	IE	0404	1028	CYRILLIC CAPITAL LETTER UKRAINIAN IE
S	DS	0405	1029	CYRILLIC CAPITAL LETTER DZE
І	II	0406	1030	CYRILLIC CAPITAL LETTER BYELORUSSIAN-UKRAINIAN I
Ї	YI	0407	1031	CYRILLIC CAPITAL LETTER YI
J	J%	0408	1032	CYRILLIC CAPITAL LETTER JE
Љ	LJ	0409	1033	CYRILLIC CAPITAL LETTER LJE
Њ	NJ	040A	1034	CYRILLIC CAPITAL LETTER NJE
Ћ	Ts	040B	1035	CYRILLIC CAPITAL LETTER TSHE
Ќ	KJ	040C	1036	CYRILLIC CAPITAL LETTER KJE
Ў	V%	040E	1038	CYRILLIC CAPITAL LETTER SHORT U
Џ	DZ	040F	1039	CYRILLIC CAPITAL LETTER DZHE
А	A=	0410	1040	CYRILLIC CAPITAL LETTER A
Б	B=	0411	1041	CYRILLIC CAPITAL LETTER BE
В	V=	0412	1042	CYRILLIC CAPITAL LETTER VE
Г	G=	0413	1043	CYRILLIC CAPITAL LETTER GHE
Д	D=	0414	1044	CYRILLIC CAPITAL LETTER DE
Е	E=	0415	1045	CYRILLIC CAPITAL LETTER IE

Ж	Z%	0416	1046	CYRILLIC CAPITAL LETTER ZHE
З	Z=	0417	1047	CYRILLIC CAPITAL LETTER ZE
И	I=	0418	1048	CYRILLIC CAPITAL LETTER I
Й	J=	0419	1049	CYRILLIC CAPITAL LETTER SHORT I
К	K=	041A	1050	CYRILLIC CAPITAL LETTER KA
Л	L=	041B	1051	CYRILLIC CAPITAL LETTER EL
М	M=	041C	1052	CYRILLIC CAPITAL LETTER EM
Н	N=	041D	1053	CYRILLIC CAPITAL LETTER EN
О	O=	041E	1054	CYRILLIC CAPITAL LETTER O
П	P=	041F	1055	CYRILLIC CAPITAL LETTER PE
Р	R=	0420	1056	CYRILLIC CAPITAL LETTER ER
С	S=	0421	1057	CYRILLIC CAPITAL LETTER ES
Т	T=	0422	1058	CYRILLIC CAPITAL LETTER TE
У	U=	0423	1059	CYRILLIC CAPITAL LETTER U
Ф	F=	0424	1060	CYRILLIC CAPITAL LETTER EF
Х	H=	0425	1061	CYRILLIC CAPITAL LETTER HA
Ц	C=	0426	1062	CYRILLIC CAPITAL LETTER TSE
Ч	C%	0427	1063	CYRILLIC CAPITAL LETTER CHE
Ш	S%	0428	1064	CYRILLIC CAPITAL LETTER SHA
Щ	Sc	0429	1065	CYRILLIC CAPITAL LETTER SHCHA
Ъ	="	042A	1066	CYRILLIC CAPITAL LETTER HARD SIGN
Ы	Y=	042B	1067	CYRILLIC CAPITAL LETTER YERU
Ь	%"	042C	1068	CYRILLIC CAPITAL LETTER SOFT SIGN
Э	JE	042D	1069	CYRILLIC CAPITAL LETTER E
Ю	JU	042E	1070	CYRILLIC CAPITAL LETTER YU
Я	JA	042F	1071	CYRILLIC CAPITAL LETTER YA
а	a=	0430	1072	CYRILLIC SMALL LETTER A
б	b=	0431	1073	CYRILLIC SMALL LETTER BE
в	v=	0432	1074	CYRILLIC SMALL LETTER VE
г	g=	0433	1075	CYRILLIC SMALL LETTER GHE
д	d=	0434	1076	CYRILLIC SMALL LETTER DE
е	e=	0435	1077	CYRILLIC SMALL LETTER IE
ж	z%	0436	1078	CYRILLIC SMALL LETTER ZHE
з	z=	0437	1079	CYRILLIC SMALL LETTER ZE
и	i=	0438	1080	CYRILLIC SMALL LETTER I
й	j=	0439	1081	CYRILLIC SMALL LETTER SHORT I
к	k=	043A	1082	CYRILLIC SMALL LETTER KA
л	l=	043B	1083	CYRILLIC SMALL LETTER EL
м	m=	043C	1084	CYRILLIC SMALL LETTER EM
н	n=	043D	1085	CYRILLIC SMALL LETTER EN
о	o=	043E	1086	CYRILLIC SMALL LETTER O
п	p=	043F	1087	CYRILLIC SMALL LETTER PE
р	r=	0440	1088	CYRILLIC SMALL LETTER ER
с	s=	0441	1089	CYRILLIC SMALL LETTER ES
т	t=	0442	1090	CYRILLIC SMALL LETTER TE
у	u=	0443	1091	CYRILLIC SMALL LETTER U
ф	f=	0444	1092	CYRILLIC SMALL LETTER EF
х	h=	0445	1093	CYRILLIC SMALL LETTER HA
ц	c=	0446	1094	CYRILLIC SMALL LETTER TSE
ч	c%	0447	1095	CYRILLIC SMALL LETTER CHE
ш	s%	0448	1096	CYRILLIC SMALL LETTER SHA
щ	sc	0449	1097	CYRILLIC SMALL LETTER SHCHA
ъ	='	044A	1098	CYRILLIC SMALL LETTER HARD SIGN
ы	y=	044B	1099	CYRILLIC SMALL LETTER YERU
ь	%'	044C	1100	CYRILLIC SMALL LETTER SOFT SIGN
э	je	044D	1101	CYRILLIC SMALL LETTER E
ю	ju	044E	1102	CYRILLIC SMALL LETTER YU
я	ja	044F	1103	CYRILLIC SMALL LETTER YA
ё	io	0451	1105	CYRILLIC SMALL LETTER IO
ђ	d%	0452	1106	CYRILLIC SMALL LETTER DJE
ѓ	g%	0453	1107	CYRILLIC SMALL LETTER GJE
є	ie	0454	1108	CYRILLIC SMALL LETTER UKRAINIAN IE
ѕ	ds	0455	1109	CYRILLIC SMALL LETTER DZE
і	ii	0456	1110	CYRILLIC SMALL LETTER BYELORUSSIAN-UKRAINIAN I
ї	yi	0457	1111	CYRILLIC SMALL LETTER YI
ј	j%	0458	1112	CYRILLIC SMALL LETTER JE
љ	lj	0459	1113	CYRILLIC SMALL LETTER LJE
њ	nj	045A	1114	CYRILLIC SMALL LETTER NJE
ћ	ts	045B	1115	CYRILLIC SMALL LETTER TSHE
ќ	kj	045C	1116	CYRILLIC SMALL LETTER KJE
ў	v%	045E	1118	CYRILLIC SMALL LETTER SHORT U
џ	dz	045F	1119	CYRILLIC SMALL LETTER DZHE
Ѣ	Y3	0462	1122	CYRILLIC CAPITAL LETTER YAT
ѣ	y3	0463	1123	CYRILLIC SMALL LETTER YAT
Ѫ	O3	046A	1130	CYRILLIC CAPITAL LETTER BIG YUS
ѫ	o3	046B	1131	CYRILLIC SMALL LETTER BIG YUS
Ѳ	F3	0472	1138	CYRILLIC CAPITAL LETTER FITA
ѳ	f3	0473	1139	CYRILLIC SMALL LETTER FITA
Ѵ	V3	0474	1140	CYRILLIC CAPITAL LETTER IZHITSA
ѵ	v3	0475	1141	CYRILLIC SMALL LETTER IZHITSA
Ҁ	C3	0480	1152	CYRILLIC CAPITAL LETTER KOPPA
ҁ	c3	0481	1153	CYRILLIC SMALL LETTER KOPPA
Ґ	G3	0490	1168	CYRILLIC CAPITAL LETTER GHE WITH UPTURN
ґ	g3	0491	1169	CYRILLIC SMALL LETTER GHE WITH UPTURN
א	A+	05D0	1488	HEBREW LETTER ALEF
ב	B+	05D1	1489	HEBREW LETTER BET
ג	G+	05D2	1490	HEBREW LETTER GIMEL
ד	D+	05D3	1491	HEBREW LETTER DALET
ה	H+	05D4	1492	HEBREW LETTER HE
ו	W+	05D5	1493	HEBREW LETTER VAV

ז	Z+	05D6	1494	HEBREW LETTER ZAYIN
ח	X+	05D7	1495	HEBREW LETTER HET
ט	Tj	05D8	1496	HEBREW LETTER TET
י	J+	05D9	1497	HEBREW LETTER YOD
ך	K%	05DA	1498	HEBREW LETTER FINAL KAF
כ	K+	05DB	1499	HEBREW LETTER KAF
ל	L+	05DC	1500	HEBREW LETTER LAMED
ם	M%	05DD	1501	HEBREW LETTER FINAL MEM
מ	M+	05DE	1502	HEBREW LETTER MEM
ן	N%	05DF	1503	HEBREW LETTER FINAL NUN `
נ	N+	05E0	1504	HEBREW LETTER NUN `
ס	S+	05E1	1505	HEBREW LETTER SAMEKH
ע	E+	05E2	1506	HEBREW LETTER AYIN
ף	P%	05E3	1507	HEBREW LETTER FINAL PE
פ	P+	05E4	1508	HEBREW LETTER PE
ץ	Zj	05E5	1509	HEBREW LETTER FINAL TSADI
צ	ZJ	05E6	1510	HEBREW LETTER TSADI
ק	Q+	05E7	1511	HEBREW LETTER QOF
ר	R+	05E8	1512	HEBREW LETTER RESH
ש	Sh	05E9	1513	HEBREW LETTER SHIN
ת	T+	05EA	1514	HEBREW LETTER TAV
،	,+	060C	1548	ARABIC COMMA
؛	+;	061B	1563	ARABIC SEMICOLON
؟	+?	061F	1567	ARABIC QUESTION MARK
ء	H'	0621	1569	ARABIC LETTER HAMZA
آ	aM	0622	1570	ARABIC LETTER ALEF WITH MADDA ABOVE
أ	aH	0623	1571	ARABIC LETTER ALEF WITH HAMZA ABOVE
ؤ	wH	0624	1572	ARABIC LETTER WAW WITH HAMZA ABOVE
إ	ah	0625	1573	ARABIC LETTER ALEF WITH HAMZA BELOW
ئ	yH	0626	1574	ARABIC LETTER YEH WITH HAMZA ABOVE
ا	a+	0627	1575	ARABIC LETTER ALEF
ب	b+	0628	1576	ARABIC LETTER BEH
ة	tm	0629	1577	ARABIC LETTER TEH MARBUTA
ت	t+	062A	1578	ARABIC LETTER TEH
ث	tk	062B	1579	ARABIC LETTER THEH
ج	g+	062C	1580	ARABIC LETTER JEEM
ح	hk	062D	1581	ARABIC LETTER HAH
خ	x+	062E	1582	ARABIC LETTER KHAH
د	d+	062F	1583	ARABIC LETTER DAL
ذ	dk	0630	1584	ARABIC LETTER THAL
ر	r+	0631	1585	ARABIC LETTER REH
ز	z+	0632	1586	ARABIC LETTER ZAIN
س	s+	0633	1587	ARABIC LETTER SEEN
ش	sn	0634	1588	ARABIC LETTER SHEEN
ص	c+	0635	1589	ARABIC LETTER SAD
ض	dd	0636	1590	ARABIC LETTER DAD
ط	tj	0637	1591	ARABIC LETTER TAH
ظ	zH	0638	1592	ARABIC LETTER ZAH
ع	e+	0639	1593	ARABIC LETTER AIN
غ	i+	063A	1594	ARABIC LETTER GHAIN
ـ	++	0640	1600	ARABIC TATWEEL
ف	f+	0641	1601	ARABIC LETTER FEH
ق	q+	0642	1602	ARABIC LETTER QAF
ك	k+	0643	1603	ARABIC LETTER KAF
ل	l+	0644	1604	ARABIC LETTER LAM
م	m+	0645	1605	ARABIC LETTER MEEM
ن	n+	0646	1606	ARABIC LETTER NOON
ه	h+	0647	1607	ARABIC LETTER HEH
و	w+	0648	1608	ARABIC LETTER WAW
ى	j+	0649	1609	ARABIC LETTER ALEF MAKSURA
ي	y+	064A	1610	ARABIC LETTER YEH
ً	:+	064B	1611	ARABIC FATHATAN
ٌ	"+	064C	1612	ARABIC DAMMATAN
ٍ	=+	064D	1613	ARABIC KASRATAN
َ	/+	064E	1614	ARABIC FATHA
ُ	'+	064F	1615	ARABIC DAMMA
ِ	1+	0650	1616	ARABIC KASRA
ّ	3+	0651	1617	ARABIC SHADDA
ْ	0+	0652	1618	ARABIC SUKUN
ٰ	aS	0670	1648	ARABIC LETTER SUPERSCRIPT ALEF
پ	p+	067E	1662	ARABIC LETTER PEH
ڤ	v+	06A4	1700	ARABIC LETTER VEH
گ	gf	06AF	1711	ARABIC LETTER GAF
۰	0a	06F0	1776	EXTENDED ARABIC-INDIC DIGIT ZERO
۱	1a	06F1	1777	EXTENDED ARABIC-INDIC DIGIT ONE
۲	2a	06F2	1778	EXTENDED ARABIC-INDIC DIGIT TWO
۳	3a	06F3	1779	EXTENDED ARABIC-INDIC DIGIT THREE
۴	4a	06F4	1780	EXTENDED ARABIC-INDIC DIGIT FOUR
۵	5a	06F5	1781	EXTENDED ARABIC-INDIC DIGIT FIVE
۶	6a	06F6	1782	EXTENDED ARABIC-INDIC DIGIT SIX
۷	7a	06F7	1783	EXTENDED ARABIC-INDIC DIGIT SEVEN
۸	8a	06F8	1784	EXTENDED ARABIC-INDIC DIGIT EIGHT
۹	9a	06F9	1785	EXTENDED ARABIC-INDIC DIGIT NINE
Ḃ	B.	1E02	7682	LATIN CAPITAL LETTER B WITH DOT ABOVE
ḃ	b.	1E03	7683	LATIN SMALL LETTER B WITH DOT ABOVE
Ḇ	B_	1E06	7686	LATIN CAPITAL LETTER B WITH LINE BELOW
ḇ	b_	1E07	7687	LATIN SMALL LETTER B WITH LINE BELOW
Ḋ	D.	1E0A	7690	LATIN CAPITAL LETTER D WITH DOT ABOVE
ḋ	d.	1E0B	7691	LATIN SMALL LETTER D WITH DOT ABOVE
Ḏ	D_	1E0E	7694	LATIN CAPITAL LETTER D WITH LINE BELOW

ḏ	d_	1E0F	7695	LATIN SMALL LETTER D WITH LINE BELOW
Ḑ	D,	1E10	7696	LATIN CAPITAL LETTER D WITH CEDILLA
ḑ	d,	1E11	7697	LATIN SMALL LETTER D WITH CEDILLA
Ḟ	F.	1E1E	7710	LATIN CAPITAL LETTER F WITH DOT ABOVE
ḟ	f.	1E1F	7711	LATIN SMALL LETTER F WITH DOT ABOVE
Ḡ	G-	1E20	7712	LATIN CAPITAL LETTER G WITH MACRON
ḡ	g-	1E21	7713	LATIN SMALL LETTER G WITH MACRON
Ḣ	H.	1E22	7714	LATIN CAPITAL LETTER H WITH DOT ABOVE
ḣ	h.	1E23	7715	LATIN SMALL LETTER H WITH DOT ABOVE
Ḧ	H:	1E26	7718	LATIN CAPITAL LETTER H WITH DIAERESIS
ḧ	h:	1E27	7719	LATIN SMALL LETTER H WITH DIAERESIS
Ḩ	H,	1E28	7720	LATIN CAPITAL LETTER H WITH CEDILLA
ḩ	h,	1E29	7721	LATIN SMALL LETTER H WITH CEDILLA
Ḱ	K'	1E30	7728	LATIN CAPITAL LETTER K WITH ACUTE
ḱ	k'	1E31	7729	LATIN SMALL LETTER K WITH ACUTE
Ḵ	K_	1E34	7732	LATIN CAPITAL LETTER K WITH LINE BELOW
ḵ	k_	1E35	7733	LATIN SMALL LETTER K WITH LINE BELOW
Ḻ	L_	1E3A	7738	LATIN CAPITAL LETTER L WITH LINE BELOW
ḻ	l_	1E3B	7739	LATIN SMALL LETTER L WITH LINE BELOW
Ḿ	M'	1E3E	7742	LATIN CAPITAL LETTER M WITH ACUTE
ḿ	m'	1E3F	7743	LATIN SMALL LETTER M WITH ACUTE
Ṁ	M.	1E40	7744	LATIN CAPITAL LETTER M WITH DOT ABOVE
ṁ	m.	1E41	7745	LATIN SMALL LETTER M WITH DOT ABOVE
Ṅ	N.	1E44	7748	LATIN CAPITAL LETTER N WITH DOT ABOVE `
ṅ	n.	1E45	7749	LATIN SMALL LETTER N WITH DOT ABOVE `
Ṉ	N_	1E48	7752	LATIN CAPITAL LETTER N WITH LINE BELOW `
ṉ	n_	1E49	7753	LATIN SMALL LETTER N WITH LINE BELOW `
Ṕ	P'	1E54	7764	LATIN CAPITAL LETTER P WITH ACUTE
ṕ	p'	1E55	7765	LATIN SMALL LETTER P WITH ACUTE
Ṗ	P.	1E56	7766	LATIN CAPITAL LETTER P WITH DOT ABOVE
ṗ	p.	1E57	7767	LATIN SMALL LETTER P WITH DOT ABOVE
Ṙ	R.	1E58	7768	LATIN CAPITAL LETTER R WITH DOT ABOVE
ṙ	r.	1E59	7769	LATIN SMALL LETTER R WITH DOT ABOVE
Ṟ	R_	1E5E	7774	LATIN CAPITAL LETTER R WITH LINE BELOW
ṟ	r_	1E5F	7775	LATIN SMALL LETTER R WITH LINE BELOW
Ṡ	S.	1E60	7776	LATIN CAPITAL LETTER S WITH DOT ABOVE
ṡ	s.	1E61	7777	LATIN SMALL LETTER S WITH DOT ABOVE
Ṫ	T.	1E6A	7786	LATIN CAPITAL LETTER T WITH DOT ABOVE
ṫ	t.	1E6B	7787	LATIN SMALL LETTER T WITH DOT ABOVE
Ṯ	T_	1E6E	7790	LATIN CAPITAL LETTER T WITH LINE BELOW
ṯ	t_	1E6F	7791	LATIN SMALL LETTER T WITH LINE BELOW
Ṽ	V?	1E7C	7804	LATIN CAPITAL LETTER V WITH TILDE
ṽ	v?	1E7D	7805	LATIN SMALL LETTER V WITH TILDE
Ẁ	W!	1E80	7808	LATIN CAPITAL LETTER W WITH GRAVE
ẁ	w!	1E81	7809	LATIN SMALL LETTER W WITH GRAVE
Ẃ	W'	1E82	7810	LATIN CAPITAL LETTER W WITH ACUTE
ẃ	w'	1E83	7811	LATIN SMALL LETTER W WITH ACUTE
Ẅ	W:	1E84	7812	LATIN CAPITAL LETTER W WITH DIAERESIS
ẅ	w:	1E85	7813	LATIN SMALL LETTER W WITH DIAERESIS
Ẇ	W.	1E86	7814	LATIN CAPITAL LETTER W WITH DOT ABOVE
ẇ	w.	1E87	7815	LATIN SMALL LETTER W WITH DOT ABOVE
Ẋ	X.	1E8A	7818	LATIN CAPITAL LETTER X WITH DOT ABOVE
ẋ	x.	1E8B	7819	LATIN SMALL LETTER X WITH DOT ABOVE
Ẍ	X:	1E8C	7820	LATIN CAPITAL LETTER X WITH DIAERESIS
ẍ	x:	1E8D	7821	LATIN SMALL LETTER X WITH DIAERESIS
Ẏ	Y.	1E8E	7822	LATIN CAPITAL LETTER Y WITH DOT ABOVE
ẏ	y.	1E8F	7823	LATIN SMALL LETTER Y WITH DOT ABOVE
Ẑ	Z>	1E90	7824	LATIN CAPITAL LETTER Z WITH CIRCUMFLEX
ẑ	z>	1E91	7825	LATIN SMALL LETTER Z WITH CIRCUMFLEX
Ẓ	Z_	1E94	7828	LATIN CAPITAL LETTER Z WITH LINE BELOW
ẓ	z_	1E95	7829	LATIN SMALL LETTER Z WITH LINE BELOW
ẖ	h_	1E96	7830	LATIN SMALL LETTER H WITH LINE BELOW
ẗ	t:	1E97	7831	LATIN SMALL LETTER T WITH DIAERESIS
ẘ	w0	1E98	7832	LATIN SMALL LETTER W WITH RING ABOVE
ẙ	y0	1E99	7833	LATIN SMALL LETTER Y WITH RING ABOVE
Ả	A2	1EA2	7842	LATIN CAPITAL LETTER A WITH HOOK ABOVE
ả	a2	1EA3	7843	LATIN SMALL LETTER A WITH HOOK ABOVE
Ẻ	E2	1EBA	7866	LATIN CAPITAL LETTER E WITH HOOK ABOVE
ẻ	e2	1EBB	7867	LATIN SMALL LETTER E WITH HOOK ABOVE
Ẽ	E?	1EBC	7868	LATIN CAPITAL LETTER E WITH TILDE
ẽ	e?	1EBD	7869	LATIN SMALL LETTER E WITH TILDE
Ỉ	I2	1EC8	7880	LATIN CAPITAL LETTER I WITH HOOK ABOVE
ỉ	i2	1EC9	7881	LATIN SMALL LETTER I WITH HOOK ABOVE
Ỏ	O2	1ECE	7886	LATIN CAPITAL LETTER O WITH HOOK ABOVE
ỏ	o2	1ECF	7887	LATIN SMALL LETTER O WITH HOOK ABOVE
Ủ	U2	1EE6	7910	LATIN CAPITAL LETTER U WITH HOOK ABOVE
ủ	u2	1EE7	7911	LATIN SMALL LETTER U WITH HOOK ABOVE
Ỳ	Y!	1EF2	7922	LATIN CAPITAL LETTER Y WITH GRAVE
ỳ	y!	1EF3	7923	LATIN SMALL LETTER Y WITH GRAVE
Ỷ	Y2	1EF6	7926	LATIN CAPITAL LETTER Y WITH HOOK ABOVE
ỷ	y2	1EF7	7927	LATIN SMALL LETTER Y WITH HOOK ABOVE
Ỹ	Y?	1EF8	7928	LATIN CAPITAL LETTER Y WITH TILDE
ỹ	y?	1EF9	7929	LATIN SMALL LETTER Y WITH TILDE
ἀ	;'	1F00	7936	GREEK SMALL LETTER ALPHA WITH PSILI
ἁ	,'	1F01	7937	GREEK SMALL LETTER ALPHA WITH DASIA
ἂ	;!	1F02	7938	GREEK SMALL LETTER ALPHA WITH PSILI AND VARIA
ἃ	,!	1F03	7939	GREEK SMALL LETTER ALPHA WITH DASIA AND VARIA
ἄ	?;	1F04	7940	GREEK SMALL LETTER ALPHA WITH PSILI AND OXIA
ἅ	?,	1F05	7941	GREEK SMALL LETTER ALPHA WITH DASIA AND OXIA
ἆ	!:	1F06	7942	GREEK SMALL LETTER ALPHA WITH PSILI AND PERISPOMENI

ᾷ	?:	1F07	7943	GREEK SMALL LETTER ALPHA WITH DASIA AND PERISPOMENI
	1N	2002	8194	EN SPACE
	1M	2003	8195	EM SPACE
	3M	2004	8196	THREE-PER-EM SPACE
	4M	2005	8197	FOUR-PER-EM SPACE
	6M	2006	8198	SIX-PER-EM SPACE
	1T	2009	8201	THIN SPACE
	1H	200A	8202	HAIR SPACE
‐	-1	2010	8208	HYPHEN
–	-N	2013	8211	EN DASH `
—	-M	2014	8212	EM DASH
―	-3	2015	8213	HORIZONTAL BAR
‖	!2	2016	8214	DOUBLE VERTICAL LINE
‗	=2	2017	8215	DOUBLE LOW LINE
‘	'6	2018	8216	LEFT SINGLE QUOTATION MARK
’	'9	2019	8217	RIGHT SINGLE QUOTATION MARK
‚	.9	201A	8218	SINGLE LOW-9 QUOTATION MARK
‛	9'	201B	8219	SINGLE HIGH-REVERSED-9 QUOTATION MARK
“	"6	201C	8220	LEFT DOUBLE QUOTATION MARK
”	"9	201D	8221	RIGHT DOUBLE QUOTATION MARK
„	:9	201E	8222	DOUBLE LOW-9 QUOTATION MARK
‟	9"	201F	8223	DOUBLE HIGH-REVERSED-9 QUOTATION MARK
†	/-	2020	8224	DAGGER
‡	/=	2021	8225	DOUBLE DAGGER
•	oo	2022	8226	BULLET
‥	..	2025	8229	TWO DOT LEADER
…	,.	2026	8230	HORIZONTAL ELLIPSIS
‰	%0	2030	8240	PER MILLE SIGN
′	1'	2032	8242	PRIME
″	2'	2033	8243	DOUBLE PRIME
‴	3'	2034	8244	TRIPLE PRIME
⁗	4'	2057	8279	QUADRUPLE PRIME
‵	1"	2035	8245	REVERSED PRIME
‶	2"	2036	8246	REVERSED DOUBLE PRIME
‷	3"	2037	8247	REVERSED TRIPLE PRIME
‸	Ca	2038	8248	CARET
‹	<1	2039	8249	SINGLE LEFT-POINTING ANGLE QUOTATION MARK
›	>1	203A	8250	SINGLE RIGHT-POINTING ANGLE QUOTATION MARK
※	:X	203B	8251	REFERENCE MARK
‾	'-	203E	8254	OVERLINE
⁄	/f	2044	8260	FRACTION SLASH
⁰	0S	2070	8304	SUPERSCRIPT ZERO
⁴	4S	2074	8308	SUPERSCRIPT FOUR
⁵	5S	2075	8309	SUPERSCRIPT FIVE
⁶	6S	2076	8310	SUPERSCRIPT SIX
⁷	7S	2077	8311	SUPERSCRIPT SEVEN
⁸	8S	2078	8312	SUPERSCRIPT EIGHT
⁹	9S	2079	8313	SUPERSCRIPT NINE
⁺	+S	207A	8314	SUPERSCRIPT PLUS SIGN
⁻	-S	207B	8315	SUPERSCRIPT MINUS
⁼	=S	207C	8316	SUPERSCRIPT EQUALS SIGN
⁽	(S	207D	8317	SUPERSCRIPT LEFT PARENTHESIS
⁾	)S	207E	8318	SUPERSCRIPT RIGHT PARENTHESIS
ⁿ	nS	207F	8319	SUPERSCRIPT LATIN SMALL LETTER N `
₀	0s	2080	8320	SUBSCRIPT ZERO
₁	1s	2081	8321	SUBSCRIPT ONE
₂	2s	2082	8322	SUBSCRIPT TWO
₃	3s	2083	8323	SUBSCRIPT THREE
₄	4s	2084	8324	SUBSCRIPT FOUR
₅	5s	2085	8325	SUBSCRIPT FIVE
₆	6s	2086	8326	SUBSCRIPT SIX
₇	7s	2087	8327	SUBSCRIPT SEVEN
₈	8s	2088	8328	SUBSCRIPT EIGHT
₉	9s	2089	8329	SUBSCRIPT NINE
₊	+s	208A	8330	SUBSCRIPT PLUS SIGN
₋	-s	208B	8331	SUBSCRIPT MINUS
₌	=s	208C	8332	SUBSCRIPT EQUALS SIGN
₍	(s	208D	8333	SUBSCRIPT LEFT PARENTHESIS
₎	)s	208E	8334	SUBSCRIPT RIGHT PARENTHESIS
£	Li	20A4	8356	LIRA SIGN
₧	Pt	20A7	8359	PESETA SIGN
₩	W=	20A9	8361	WON SIGN
€	Eu	20AC	8364	EURO SIGN
₽	=R	20BD	8381	ROUBLE SIGN
₽	=P	20BD	8381	ROUBLE SIGN
℃	oC	2103	8451	DEGREE CELSIUS
℅	co	2105	8453	CARE OF
℉	oF	2109	8457	DEGREE FAHRENHEIT
№	N0	2116	8470	NUMERO SIGN
℗	PO	2117	8471	SOUND RECORDING COPYRIGHT
℞	Rx	211E	8478	PRESCRIPTION TAKE
℠	SM	2120	8480	SERVICE MARK
™	TM	2122	8482	TRADE MARK SIGN
Ω	Om	2126	8486	OHM SIGN
Å	AO	212B	8491	ANGSTROM SIGN
⅓	13	2153	8531	VULGAR FRACTION ONE THIRD
⅔	23	2154	8532	VULGAR FRACTION TWO THIRDS
⅕	15	2155	8533	VULGAR FRACTION ONE FIFTH
⅖	25	2156	8534	VULGAR FRACTION TWO FIFTHS
⅗	35	2157	8535	VULGAR FRACTION THREE FIFTHS

⅘	45	2158	8536	VULGAR FRACTION FOUR FIFTHS
⅙	16	2159	8537	VULGAR FRACTION ONE SIXTH
⅚	56	215A	8538	VULGAR FRACTION FIVE SIXTHS
⅛	18	215B	8539	VULGAR FRACTION ONE EIGHTH
⅜	38	215C	8540	VULGAR FRACTION THREE EIGHTHS
⅝	58	215D	8541	VULGAR FRACTION FIVE EIGHTHS
⅞	78	215E	8542	VULGAR FRACTION SEVEN EIGHTHS
Ⅰ	1R	2160	8544	ROMAN NUMERAL ONE
Ⅱ	2R	2161	8545	ROMAN NUMERAL TWO
Ⅲ	3R	2162	8546	ROMAN NUMERAL THREE
Ⅳ	4R	2163	8547	ROMAN NUMERAL FOUR
Ⅴ	5R	2164	8548	ROMAN NUMERAL FIVE
Ⅵ	6R	2165	8549	ROMAN NUMERAL SIX
Ⅶ	7R	2166	8550	ROMAN NUMERAL SEVEN
Ⅷ	8R	2167	8551	ROMAN NUMERAL EIGHT
Ⅸ	9R	2168	8552	ROMAN NUMERAL NINE
Ⅹ	aR	2169	8553	ROMAN NUMERAL TEN
Ⅺ	bR	216A	8554	ROMAN NUMERAL ELEVEN
Ⅻ	cR	216B	8555	ROMAN NUMERAL TWELVE
ⅰ	1r	2170	8560	SMALL ROMAN NUMERAL ONE
ⅱ	2r	2171	8561	SMALL ROMAN NUMERAL TWO
ⅲ	3r	2172	8562	SMALL ROMAN NUMERAL THREE
ⅳ	4r	2173	8563	SMALL ROMAN NUMERAL FOUR
ⅴ	5r	2174	8564	SMALL ROMAN NUMERAL FIVE
ⅵ	6r	2175	8565	SMALL ROMAN NUMERAL SIX
ⅶ	7r	2176	8566	SMALL ROMAN NUMERAL SEVEN
ⅷ	8r	2177	8567	SMALL ROMAN NUMERAL EIGHT
ⅸ	9r	2178	8568	SMALL ROMAN NUMERAL NINE
ⅹ	ar	2179	8569	SMALL ROMAN NUMERAL TEN
ⅺ	br	217A	8570	SMALL ROMAN NUMERAL ELEVEN
ⅻ	cr	217B	8571	SMALL ROMAN NUMERAL TWELVE
←	<-	2190	8592	LEFTWARDS ARROW
↑	-!	2191	8593	UPWARDS ARROW
→	->	2192	8594	RIGHTWARDS ARROW
↓	-v	2193	8595	DOWNWARDS ARROW
↔	<>	2194	8596	LEFT RIGHT ARROW
↕	UD	2195	8597	UP DOWN ARROW
⇐	<=	21D0	8656	LEFTWARDS DOUBLE ARROW
⇒	=>	21D2	8658	RIGHTWARDS DOUBLE ARROW
⇔	==	21D4	8660	LEFT RIGHT DOUBLE ARROW
∀	FA	2200	8704	FOR ALL
∂	dP	2202	8706	PARTIAL DIFFERENTIAL
∃	TE	2203	8707	THERE EXISTS
∅	/0	2205	8709	EMPTY SET
∆	DE	2206	8710	INCREMENT
∇	NB	2207	8711	NABLA
∈	(-	2208	8712	ELEMENT OF
∋	-)	220B	8715	CONTAINS AS MEMBER
∏	*P	220F	8719	N-ARY PRODUCT `
∑	+Z	2211	8721	N-ARY SUMMATION `
−	-2	2212	8722	MINUS SIGN
∓	-+	2213	8723	MINUS-OR-PLUS SIGN
∗	*-	2217	8727	ASTERISK OPERATOR
∘	Ob	2218	8728	RING OPERATOR
•	Sb	2219	8729	BULLET OPERATOR
√	RT	221A	8730	SQUARE ROOT
∝	0(	221D	8733	PROPORTIONAL TO
∞	00	221E	8734	INFINITY
∟	-L	221F	8735	RIGHT ANGLE
∠	-V	2220	8736	ANGLE
∥	PP	2225	8741	PARALLEL TO
∧	AN	2227	8743	LOGICAL AND
∨	OR	2228	8744	LOGICAL OR
∩	(U	2229	8745	INTERSECTION
∪	)U	222A	8746	UNION
∫	In	222B	8747	INTEGRAL
∬	DI	222C	8748	DOUBLE INTEGRAL
∮	Io	222E	8750	CONTOUR INTEGRAL
∴	.:	2234	8756	THEREFORE
∵	:.	2235	8757	BECAUSE
∶	:R	2236	8758	RATIO
∷	::	2237	8759	PROPORTION
~	?1	223C	8764	TILDE OPERATOR
∾	CG	223E	8766	INVERTED LAZY S
≃	?-	2243	8771	ASYMPTOTICALLY EQUAL TO
≅	?=	2245	8773	APPROXIMATELY EQUAL TO
≈	?2	2248	8776	ALMOST EQUAL TO
≡	=?	224C	8780	ALL EQUAL TO
≐	.=	2250	8784	APPROACHES THE LIMIT
≒	HI	2253	8787	IMAGE OF OR APPROXIMATELY EQUAL TO
≠	!=	2260	8800	NOT EQUAL TO
≡	=3	2261	8801	IDENTICAL TO
≤	=<	2264	8804	LESS-THAN OR EQUAL TO
≥	>=	2265	8805	GREATER-THAN OR EQUAL TO
≪	<*	226A	8810	MUCH LESS-THAN
≫	*>	226B	8811	MUCH GREATER-THAN
≮	!<	226E	8814	NOT LESS-THAN
≯	!>	226F	8815	NOT GREATER-THAN
⊂	(C	2282	8834	SUBSET OF
⊃	)C	2283	8835	SUPERSET OF

⊆	(_	2286	8838	SUBSET OF OR EQUAL TO
⊇	)_	2287	8839	SUPERSET OF OR EQUAL TO
⊙	0.	2299	8857	CIRCLED DOT OPERATOR
⊚	02	229A	8858	CIRCLED RING OPERATOR
⊥	-T	22A5	8869	UP TACK
⋅	.P	22C5	8901	DOT OPERATOR
⋮	:3	22EE	8942	VERTICAL ELLIPSIS
⋯	.3	22EF	8943	MIDLINE HORIZONTAL ELLIPSIS
⌂	Eh	2302	8962	HOUSE
⌈	<7	2308	8968	LEFT CEILING
⌉	>7	2309	8969	RIGHT CEILING
⌊	7<	230A	8970	LEFT FLOOR
⌋	7>	230B	8971	RIGHT FLOOR
⌐	NI	2310	8976	REVERSED NOT SIGN
⌒	(A	2312	8978	ARC
⌕	TR	2315	8981	TELEPHONE RECORDER
⌠	Iu	2320	8992	TOP HALF INTEGRAL
⌡	Il	2321	8993	BOTTOM HALF INTEGRAL
⟨	<[	27E8	10040	LEFT MATHEMATICAL ANGLE BRACKET
⟩	]>	27E9	10041	RIGHT MATHEMATICAL ANGLE BRACKET
␣	Vs	2423	9251	OPEN BOX
␣	1h	2440	9280	OCR HOOK
␣	3h	2441	9281	OCR CHAIR
␣	2h	2442	9282	OCR FORK
␣	4h	2443	9283	OCR INVERTED FORK
␣	1j	2446	9286	OCR BRANCH BANK IDENTIFICATION
␣	2j	2447	9287	OCR AMOUNT OF CHECK
␣	3j	2448	9288	OCR DASH
␣	4j	2449	9289	OCR CUSTOMER ACCOUNT NUMBER
␣	1.	2488	9352	DIGIT ONE FULL STOP
␣	2.	2489	9353	DIGIT TWO FULL STOP
␣	3.	248A	9354	DIGIT THREE FULL STOP
␣	4.	248B	9355	DIGIT FOUR FULL STOP
␣	5.	248C	9356	DIGIT FIVE FULL STOP
␣	6.	248D	9357	DIGIT SIX FULL STOP
␣	7.	248E	9358	DIGIT SEVEN FULL STOP
␣	8.	248F	9359	DIGIT EIGHT FULL STOP
␣	9.	2490	9360	DIGIT NINE FULL STOP
─	hh	2500	9472	BOX DRAWINGS LIGHT HORIZONTAL
━	HH	2501	9473	BOX DRAWINGS HEAVY HORIZONTAL
│	vv	2502	9474	BOX DRAWINGS LIGHT VERTICAL
┃	VV	2503	9475	BOX DRAWINGS HEAVY VERTICAL
┄	3-	2504	9476	BOX DRAWINGS LIGHT TRIPLE DASH HORIZONTAL
┅	3_	2505	9477	BOX DRAWINGS HEAVY TRIPLE DASH HORIZONTAL
┆	3!	2506	9478	BOX DRAWINGS LIGHT TRIPLE DASH VERTICAL
┇	3/	2507	9479	BOX DRAWINGS HEAVY TRIPLE DASH VERTICAL
┈	4-	2508	9480	BOX DRAWINGS LIGHT QUADRUPLE DASH HORIZONTAL
┉	4_	2509	9481	BOX DRAWINGS HEAVY QUADRUPLE DASH HORIZONTAL
┊	4!	250A	9482	BOX DRAWINGS LIGHT QUADRUPLE DASH VERTICAL
┋	4/	250B	9483	BOX DRAWINGS HEAVY QUADRUPLE DASH VERTICAL
┌	dr	250C	9484	BOX DRAWINGS LIGHT DOWN AND RIGHT
┍	dR	250D	9485	BOX DRAWINGS DOWN LIGHT AND RIGHT HEAVY
┎	Dr	250E	9486	BOX DRAWINGS DOWN HEAVY AND RIGHT LIGHT
┏	DR	250F	9487	BOX DRAWINGS HEAVY DOWN AND RIGHT
┐	dl	2510	9488	BOX DRAWINGS LIGHT DOWN AND LEFT
┑	dL	2511	9489	BOX DRAWINGS DOWN LIGHT AND LEFT HEAVY
┒	Dl	2512	9490	BOX DRAWINGS DOWN HEAVY AND LEFT LIGHT
┓	LD	2513	9491	BOX DRAWINGS HEAVY DOWN AND LEFT
└	ur	2514	9492	BOX DRAWINGS LIGHT UP AND RIGHT
┕	uR	2515	9493	BOX DRAWINGS UP LIGHT AND RIGHT HEAVY
┖	Ur	2516	9494	BOX DRAWINGS UP HEAVY AND RIGHT LIGHT
┗	UR	2517	9495	BOX DRAWINGS HEAVY UP AND RIGHT
┘	ul	2518	9496	BOX DRAWINGS LIGHT UP AND LEFT
┙	uL	2519	9497	BOX DRAWINGS UP LIGHT AND LEFT HEAVY
┚	Ul	251A	9498	BOX DRAWINGS UP HEAVY AND LEFT LIGHT
┛	UL	251B	9499	BOX DRAWINGS HEAVY UP AND LEFT
├	vr	251C	9500	BOX DRAWINGS LIGHT VERTICAL AND RIGHT
┝	vR	251D	9501	BOX DRAWINGS VERTICAL LIGHT AND RIGHT HEAVY
┞	Vr	2520	9504	BOX DRAWINGS VERTICAL HEAVY AND RIGHT LIGHT
┣	VR	2523	9507	BOX DRAWINGS HEAVY VERTICAL AND RIGHT
┤	vl	2524	9508	BOX DRAWINGS LIGHT VERTICAL AND LEFT
┥	vL	2525	9509	BOX DRAWINGS VERTICAL LIGHT AND LEFT HEAVY
┦	Vl	2528	9512	BOX DRAWINGS VERTICAL HEAVY AND LEFT LIGHT
┫	VL	252B	9515	BOX DRAWINGS HEAVY VERTICAL AND LEFT
┬	dh	252C	9516	BOX DRAWINGS LIGHT DOWN AND HORIZONTAL
┯	dH	252F	9519	BOX DRAWINGS DOWN LIGHT AND HORIZONTAL HEAVY
┰	Dh	2530	9520	BOX DRAWINGS DOWN HEAVY AND HORIZONTAL LIGHT
┳	DH	2533	9523	BOX DRAWINGS HEAVY DOWN AND HORIZONTAL
┴	uh	2534	9524	BOX DRAWINGS LIGHT UP AND HORIZONTAL
┷	uH	2537	9527	BOX DRAWINGS UP LIGHT AND HORIZONTAL HEAVY
┸	Uh	2538	9528	BOX DRAWINGS UP HEAVY AND HORIZONTAL LIGHT
┻	UH	253B	9531	BOX DRAWINGS HEAVY UP AND HORIZONTAL
┼	vh	253C	9532	BOX DRAWINGS LIGHT VERTICAL AND HORIZONTAL
┿	vH	253F	9535	BOX DRAWINGS VERTICAL LIGHT AND HORIZONTAL HEAVY
╂	Vh	2542	9538	BOX DRAWINGS VERTICAL HEAVY AND HORIZONTAL LIGHT
╋	VH	254B	9547	BOX DRAWINGS HEAVY VERTICAL AND HORIZONTAL
╱	FD	2571	9585	BOX DRAWINGS LIGHT DIAGONAL UPPER RIGHT TO LOWER LEFT
╲	BD	2572	9586	BOX DRAWINGS LIGHT DIAGONAL UPPER LEFT TO LOWER RIGHT
▀	TB	2580	9600	UPPER HALF BLOCK
▄	LB	2584	9604	LOWER HALF BLOCK

▉	FB	2588	9608	FULL BLOCK
▌	lB	258C	9612	LEFT HALF BLOCK
▐	RB	2590	9616	RIGHT HALF BLOCK
░	.S	2591	9617	LIGHT SHADE
▒	:S	2592	9618	MEDIUM SHADE
▓	?S	2593	9619	DARK SHADE
■	fS	25A0	9632	BLACK SQUARE
□	OS	25A1	9633	WHITE SQUARE
▢	RO	25A2	9634	WHITE SQUARE WITH ROUNDED CORNERS
▣	Rr	25A3	9635	WHITE SQUARE CONTAINING BLACK SMALL SQUARE
▤	RF	25A4	9636	SQUARE WITH HORIZONTAL FILL
▥	RY	25A5	9637	SQUARE WITH VERTICAL FILL
▦	RH	25A6	9638	SQUARE WITH ORTHOGONAL CROSSHATCH FILL
▧	RZ	25A7	9639	SQUARE WITH UPPER LEFT TO LOWER RIGHT FILL
▨	RK	25A8	9640	SQUARE WITH UPPER RIGHT TO LOWER LEFT FILL
▩	RX	25A9	9641	SQUARE WITH DIAGONAL CROSSHATCH FILL
▪	sB	25AA	9642	BLACK SMALL SQUARE
▬	SR	25AC	9644	BLACK RECTANGLE
▭	Or	25AD	9645	WHITE RECTANGLE
▲	UT	25B2	9650	BLACK UP-POINTING TRIANGLE
△	uT	25B3	9651	WHITE UP-POINTING TRIANGLE
▶	PR	25B6	9654	BLACK RIGHT-POINTING TRIANGLE
▷	Tr	25B7	9655	WHITE RIGHT-POINTING TRIANGLE
▼	Dt	25BC	9660	BLACK DOWN-POINTING TRIANGLE
▽	dT	25BD	9661	WHITE DOWN-POINTING TRIANGLE
◀	PL	25C0	9664	BLACK LEFT-POINTING TRIANGLE
◁	Tl	25C1	9665	WHITE LEFT-POINTING TRIANGLE
◆	Db	25C6	9670	BLACK DIAMOND
◇	Dw	25C7	9671	WHITE DIAMOND
◊	LZ	25CA	9674	LOZENGE
○	0m	25CB	9675	WHITE CIRCLE
◎	0o	25CE	9678	BULLSEYE
●	0M	25CF	9679	BLACK CIRCLE
◐	0L	25D0	9680	CIRCLE WITH LEFT HALF BLACK
◑	0R	25D1	9681	CIRCLE WITH RIGHT HALF BLACK
◘	Sn	25D8	9688	INVERSE BULLET
◙	Ic	25D9	9689	INVERSE WHITE CIRCLE
◢	Fd	25E2	9698	BLACK LOWER RIGHT TRIANGLE
◣	Bd	25E3	9699	BLACK LOWER LEFT TRIANGLE
★	*2	2605	9733	BLACK STAR
☆	*1	2606	9734	WHITE STAR
☜	<H	261C	9756	WHITE LEFT POINTING INDEX
☞	>H	261E	9758	WHITE RIGHT POINTING INDEX
☺	0u	263A	9786	WHITE SMILING FACE
☻	0U	263B	9787	BLACK SMILING FACE
☼	SU	263C	9788	WHITE SUN WITH RAYS
♀	Fm	2640	9792	FEMALE SIGN
♂	Ml	2642	9794	MALE SIGN
♠	cS	2660	9824	BLACK SPADE SUIT
♡	cH	2661	9825	WHITE HEART SUIT
♢	cD	2662	9826	WHITE DIAMOND SUIT
♣	cC	2663	9827	BLACK CLUB SUIT
♩	Md	2669	9833	QUARTER NOTE `
♪	M8	266A	9834	EIGHTH NOTE `
♫	M2	266B	9835	BEAMED EIGHTH NOTES
♭	Mb	266D	9837	MUSIC FLAT SIGN
♮	Mx	266E	9838	MUSIC NATURAL SIGN
♯	MX	266F	9839	MUSIC SHARP SIGN
✓	OK	2713	10003	CHECK MARK
✗	XX	2717	10007	BALLOT X
✠	-X	2720	10016	MALTESE CROSS
	IS	3000	12288	IDEOGRAPHIC SPACE
、	,_	3001	12289	IDEOGRAPHIC COMMA
。	._	3002	12290	IDEOGRAPHIC FULL STOP
〃	+"	3003	12291	DITTO MARK
〄	+_	3004	12292	JAPANESE INDUSTRIAL STANDARD SYMBOL
々	*_	3005	12293	IDEOGRAPHIC ITERATION MARK
〆	;_	3006	12294	IDEOGRAPHIC CLOSING MARK
〇	0_	3007	12295	IDEOGRAPHIC NUMBER ZERO
〈	<+	300A	12298	LEFT DOUBLE ANGLE BRACKET
〉	>+	300B	12299	RIGHT DOUBLE ANGLE BRACKET
「	<'	300C	12300	LEFT CORNER BRACKET
」	>'	300D	12301	RIGHT CORNER BRACKET
『	<"	300E	12302	LEFT WHITE CORNER BRACKET
』	>"	300F	12303	RIGHT WHITE CORNER BRACKET
【	("	3010	12304	LEFT BLACK LENTICULAR BRACKET
】	)"	3011	12305	RIGHT BLACK LENTICULAR BRACKET
〒	=T	3012	12306	POSTAL MARK
〓	=_	3013	12307	GETA MARK
〔	('	3014	12308	LEFT TORTOISE SHELL BRACKET
〕	)'	3015	12309	RIGHT TORTOISE SHELL BRACKET
〖	(I	3016	12310	LEFT WHITE LENTICULAR BRACKET
〗	)I	3017	12311	RIGHT WHITE LENTICULAR BRACKET
〜	-?	301C	12316	WAVE DASH
ぁ	A5	3041	12353	HIRAGANA LETTER SMALL A
あ	a5	3042	12354	HIRAGANA LETTER A
ぃ	I5	3043	12355	HIRAGANA LETTER SMALL I
い	i5	3044	12356	HIRAGANA LETTER I
ぅ	U5	3045	12357	HIRAGANA LETTER SMALL U
う	u5	3046	12358	HIRAGANA LETTER U

 E5 3047 12359 HIRAGANA LETTER SMALL E
 e5 3048 12360 HIRAGANA LETTER E
 O5 3049 12361 HIRAGANA LETTER SMALL O
 o5 304A 12362 HIRAGANA LETTER O
 ka 304B 12363 HIRAGANA LETTER KA
 ga 304C 12364 HIRAGANA LETTER GA
 ki 304D 12365 HIRAGANA LETTER KI
 gi 304E 12366 HIRAGANA LETTER GI
 ku 304F 12367 HIRAGANA LETTER KU
 gu 3050 12368 HIRAGANA LETTER GU
 ke 3051 12369 HIRAGANA LETTER KE
 ge 3052 12370 HIRAGANA LETTER GE
 ko 3053 12371 HIRAGANA LETTER KO
 go 3054 12372 HIRAGANA LETTER GO
 sa 3055 12373 HIRAGANA LETTER SA
 za 3056 12374 HIRAGANA LETTER ZA
 si 3057 12375 HIRAGANA LETTER SI
 zi 3058 12376 HIRAGANA LETTER ZI
 su 3059 12377 HIRAGANA LETTER SU
 zu 305A 12378 HIRAGANA LETTER ZU
 se 305B 12379 HIRAGANA LETTER SE
 ze 305C 12380 HIRAGANA LETTER ZE
 so 305D 12381 HIRAGANA LETTER SO
 zo 305E 12382 HIRAGANA LETTER ZO
 ta 305F 12383 HIRAGANA LETTER TA
 da 3060 12384 HIRAGANA LETTER DA
 ti 3061 12385 HIRAGANA LETTER TI
 di 3062 12386 HIRAGANA LETTER DI
 tU 3063 12387 HIRAGANA LETTER SMALL TU
 tu 3064 12388 HIRAGANA LETTER TU
 du 3065 12389 HIRAGANA LETTER DU
 te 3066 12390 HIRAGANA LETTER TE
 de 3067 12391 HIRAGANA LETTER DE
 to 3068 12392 HIRAGANA LETTER TO
 do 3069 12393 HIRAGANA LETTER DO
 na 306A 12394 HIRAGANA LETTER NA
 ni 306B 12395 HIRAGANA LETTER NI
 nu 306C 12396 HIRAGANA LETTER NU
 ne 306D 12397 HIRAGANA LETTER NE
 no 306E 12398 HIRAGANA LETTER NO
 ha 306F 12399 HIRAGANA LETTER HA
 ba 3070 12400 HIRAGANA LETTER BA
 pa 3071 12401 HIRAGANA LETTER PA
 hi 3072 12402 HIRAGANA LETTER HI
 bi 3073 12403 HIRAGANA LETTER BI
 pi 3074 12404 HIRAGANA LETTER PI
 hu 3075 12405 HIRAGANA LETTER HU
 bu 3076 12406 HIRAGANA LETTER BU
 pu 3077 12407 HIRAGANA LETTER PU
 he 3078 12408 HIRAGANA LETTER HE
 be 3079 12409 HIRAGANA LETTER BE
 pe 307A 12410 HIRAGANA LETTER PE
 ho 307B 12411 HIRAGANA LETTER HO
 bo 307C 12412 HIRAGANA LETTER BO
 po 307D 12413 HIRAGANA LETTER PO
 ma 307E 12414 HIRAGANA LETTER MA
 mi 307F 12415 HIRAGANA LETTER MI
 mu 3080 12416 HIRAGANA LETTER MU
 me 3081 12417 HIRAGANA LETTER ME
 mo 3082 12418 HIRAGANA LETTER MO
 yA 3083 12419 HIRAGANA LETTER SMALL YA
 ya 3084 12420 HIRAGANA LETTER YA
 yU 3085 12421 HIRAGANA LETTER SMALL YU
 yu 3086 12422 HIRAGANA LETTER YU
 yO 3087 12423 HIRAGANA LETTER SMALL YO
 yo 3088 12424 HIRAGANA LETTER YO
 ra 3089 12425 HIRAGANA LETTER RA
 ri 308A 12426 HIRAGANA LETTER RI
 ru 308B 12427 HIRAGANA LETTER RU
 re 308C 12428 HIRAGANA LETTER RE
 ro 308D 12429 HIRAGANA LETTER RO
 wA 308E 12430 HIRAGANA LETTER SMALL WA
 wa 308F 12431 HIRAGANA LETTER WA
 wi 3090 12432 HIRAGANA LETTER WI
 we 3091 12433 HIRAGANA LETTER WE
 wo 3092 12434 HIRAGANA LETTER WO
 n5 3093 12435 HIRAGANA LETTER N `
 vu 3094 12436 HIRAGANA LETTER VU
 "5 309B 12443 KATAKANA-HIRAGANA VOICED SOUND MARK
 05 309C 12444 KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK
 *5 309D 12445 HIRAGANA ITERATION MARK
 +5 309E 12446 HIRAGANA VOICED ITERATION MARK
 a6 30A1 12449 KATAKANA LETTER SMALL A
 A6 30A2 12450 KATAKANA LETTER A
 i6 30A3 12451 KATAKANA LETTER SMALL I
 I6 30A4 12452 KATAKANA LETTER I
 u6 30A5 12453 KATAKANA LETTER SMALL U
 U6 30A6 12454 KATAKANA LETTER U
 e6 30A7 12455 KATAKANA LETTER SMALL E
 E6 30A8 12456 KATAKANA LETTER E

□	o6	30A9	12457	KATAKANA LETTER SMALL O
□	O6	30AA	12458	KATAKANA LETTER O
□	Ka	30AB	12459	KATAKANA LETTER KA
□	Ga	30AC	12460	KATAKANA LETTER GA
□	Ki	30AD	12461	KATAKANA LETTER KI
□	Gi	30AE	12462	KATAKANA LETTER GI
□	Ku	30AF	12463	KATAKANA LETTER KU
□	Gu	30B0	12464	KATAKANA LETTER GU
□	Ke	30B1	12465	KATAKANA LETTER KE
□	Ge	30B2	12466	KATAKANA LETTER GE
□	Ko	30B3	12467	KATAKANA LETTER KO
□	Go	30B4	12468	KATAKANA LETTER GO
□	Sa	30B5	12469	KATAKANA LETTER SA
□	Za	30B6	12470	KATAKANA LETTER ZA
□	Si	30B7	12471	KATAKANA LETTER SI
□	Zi	30B8	12472	KATAKANA LETTER ZI
□	Su	30B9	12473	KATAKANA LETTER SU
□	Zu	30BA	12474	KATAKANA LETTER ZU
□	Se	30BB	12475	KATAKANA LETTER SE
□	Ze	30BC	12476	KATAKANA LETTER ZE
□	So	30BD	12477	KATAKANA LETTER SO
□	Zo	30BE	12478	KATAKANA LETTER ZO
□	Ta	30BF	12479	KATAKANA LETTER TA
□	Da	30C0	12480	KATAKANA LETTER DA
□	Ti	30C1	12481	KATAKANA LETTER TI
□	Di	30C2	12482	KATAKANA LETTER DI
□	TU	30C3	12483	KATAKANA LETTER SMALL TU
□	Tu	30C4	12484	KATAKANA LETTER TU
□	Du	30C5	12485	KATAKANA LETTER DU
□	Te	30C6	12486	KATAKANA LETTER TE
□	De	30C7	12487	KATAKANA LETTER DE
□	To	30C8	12488	KATAKANA LETTER TO
□	Do	30C9	12489	KATAKANA LETTER DO
□	Na	30CA	12490	KATAKANA LETTER NA
□	Ni	30CB	12491	KATAKANA LETTER NI
□	Nu	30CC	12492	KATAKANA LETTER NU
□	Ne	30CD	12493	KATAKANA LETTER NE
□	No	30CE	12494	KATAKANA LETTER NO
□	Ha	30CF	12495	KATAKANA LETTER HA
□	Ba	30D0	12496	KATAKANA LETTER BA
□	Pa	30D1	12497	KATAKANA LETTER PA
□	Hi	30D2	12498	KATAKANA LETTER HI
□	Bi	30D3	12499	KATAKANA LETTER BI
□	Pi	30D4	12500	KATAKANA LETTER PI
□	Hu	30D5	12501	KATAKANA LETTER HU
□	Bu	30D6	12502	KATAKANA LETTER BU
□	Pu	30D7	12503	KATAKANA LETTER PU
□	He	30D8	12504	KATAKANA LETTER HE
□	Be	30D9	12505	KATAKANA LETTER BE
□	Pe	30DA	12506	KATAKANA LETTER PE
□	Ho	30DB	12507	KATAKANA LETTER HO
□	Bo	30DC	12508	KATAKANA LETTER BO
□	Po	30DD	12509	KATAKANA LETTER PO
□	Ma	30DE	12510	KATAKANA LETTER MA
□	Mi	30DF	12511	KATAKANA LETTER MI
□	Mu	30E0	12512	KATAKANA LETTER MU
□	Me	30E1	12513	KATAKANA LETTER ME
□	Mo	30E2	12514	KATAKANA LETTER MO
□	YA	30E3	12515	KATAKANA LETTER SMALL YA
□	Ya	30E4	12516	KATAKANA LETTER YA
□	YU	30E5	12517	KATAKANA LETTER SMALL YU
□	Yu	30E6	12518	KATAKANA LETTER YU
□	YO	30E7	12519	KATAKANA LETTER SMALL YO
□	Yo	30E8	12520	KATAKANA LETTER YO
□	Ra	30E9	12521	KATAKANA LETTER RA
□	Ri	30EA	12522	KATAKANA LETTER RI
□	Ru	30EB	12523	KATAKANA LETTER RU
□	Re	30EC	12524	KATAKANA LETTER RE
□	Ro	30ED	12525	KATAKANA LETTER RO
□	WA	30EE	12526	KATAKANA LETTER SMALL WA
□	Wa	30EF	12527	KATAKANA LETTER WA
□	Wi	30F0	12528	KATAKANA LETTER WI
□	We	30F1	12529	KATAKANA LETTER WE
□	Wo	30F2	12530	KATAKANA LETTER WO
□	N6	30F3	12531	KATAKANA LETTER N `
□	Vu	30F4	12532	KATAKANA LETTER VU
□	KA	30F5	12533	KATAKANA LETTER SMALL KA
□	KE	30F6	12534	KATAKANA LETTER SMALL KE
□	Va	30F7	12535	KATAKANA LETTER VA
□	Vi	30F8	12536	KATAKANA LETTER VI
□	Ve	30F9	12537	KATAKANA LETTER VE
□	Vo	30FA	12538	KATAKANA LETTER VO
□	.6	30FB	12539	KATAKANA MIDDLE DOT
□	-6	30FC	12540	KATAKANA-HIRAGANA PROLONGED SOUND MARK
□	*6	30FD	12541	KATAKANA ITERATION MARK
□	+6	30FE	12542	KATAKANA VOICED ITERATION MARK
□	b4	3105	12549	BOPOMOFO LETTER B
□	p4	3106	12550	BOPOMOFO LETTER P
□	m4	3107	12551	BOPOMOFO LETTER M
□	f4	3108	12552	BOPOMOFO LETTER F

ㄉ	d4	3109	12553	BOPOMOFO LETTER D
ㄊ	t4	310A	12554	BOPOMOFO LETTER T
ㄋ	n4	310B	12555	BOPOMOFO LETTER N `
ㄌ	l4	310C	12556	BOPOMOFO LETTER L
ㄍ	g4	310D	12557	BOPOMOFO LETTER G
ㄎ	k4	310E	12558	BOPOMOFO LETTER K
ㄏ	h4	310F	12559	BOPOMOFO LETTER H
ㄐ	j4	3110	12560	BOPOMOFO LETTER J
ㄑ	q4	3111	12561	BOPOMOFO LETTER Q
ㄒ	x4	3112	12562	BOPOMOFO LETTER X
ㄓ	zh	3113	12563	BOPOMOFO LETTER ZH
ㄔ	ch	3114	12564	BOPOMOFO LETTER CH
ㄕ	sh	3115	12565	BOPOMOFO LETTER SH
ㄖ	r4	3116	12566	BOPOMOFO LETTER R
ㄗ	z4	3117	12567	BOPOMOFO LETTER Z
ㄘ	c4	3118	12568	BOPOMOFO LETTER C
ㄙ	s4	3119	12569	BOPOMOFO LETTER S
ㄚ	a4	311A	12570	BOPOMOFO LETTER A
ㄛ	o4	311B	12571	BOPOMOFO LETTER O
ㄜ	e4	311C	12572	BOPOMOFO LETTER E
ㄞ	ai	311E	12574	BOPOMOFO LETTER AI
ㄟ	ei	311F	12575	BOPOMOFO LETTER EI
ㄠ	au	3120	12576	BOPOMOFO LETTER AU
ㄡ	ou	3121	12577	BOPOMOFO LETTER OU
ㄢ	an	3122	12578	BOPOMOFO LETTER AN
ㄣ	en	3123	12579	BOPOMOFO LETTER EN
ㄤ	aN	3124	12580	BOPOMOFO LETTER ANG
ㄥ	eN	3125	12581	BOPOMOFO LETTER ENG
ㄦ	er	3126	12582	BOPOMOFO LETTER ER
ㄧ	i4	3127	12583	BOPOMOFO LETTER I
ㄨ	u4	3128	12584	BOPOMOFO LETTER U
ㄩ	iu	3129	12585	BOPOMOFO LETTER IU
ㄪ	v4	312A	12586	BOPOMOFO LETTER V
ㄫ	nG	312B	12587	BOPOMOFO LETTER NG
ㄬ	gn	312C	12588	BOPOMOFO LETTER GN
㈠	1c	3220	12832	PARENTHESIZED IDEOGRAPH ONE
㈡	2c	3221	12833	PARENTHESIZED IDEOGRAPH TWO
㈢	3c	3222	12834	PARENTHESIZED IDEOGRAPH THREE
㈣	4c	3223	12835	PARENTHESIZED IDEOGRAPH FOUR
㈤	5c	3224	12836	PARENTHESIZED IDEOGRAPH FIVE
㈥	6c	3225	12837	PARENTHESIZED IDEOGRAPH SIX
㈦	7c	3226	12838	PARENTHESIZED IDEOGRAPH SEVEN
㈧	8c	3227	12839	PARENTHESIZED IDEOGRAPH EIGHT
㈨	9c	3228	12840	PARENTHESIZED IDEOGRAPH NINE
ff	ff	FB00	64256	LATIN SMALL LIGATURE FF
fi	fi	FB01	64257	LATIN SMALL LIGATURE FI
fl	fl	FB02	64258	LATIN SMALL LIGATURE FL
ft	ft	FB05	64261	LATIN SMALL LIGATURE LONG S T
st	st	FB06	64262	LATIN SMALL LIGATURE ST

Editing files

Introduction edit-intro

Editing a file with Vim means:

1. reading the file into a buffer

2. changing the buffer with editor commands

3. writing the buffer into a file

current-file

As long as you don't write the buffer, the original file remains unchanged.
If you start editing a file (read a file into the buffer), the file name is
remembered as the "current file name". This is also known as the name of the
current buffer. It can be used with "%" on the command line :_%.

alternate-file

If there already was a current file name, then that one becomes the alternate
file name. It can be used with "#" on the command line :_# and you can use
the CTRL-^ command to toggle between the current and the alternate file.
However, the alternate file name is not changed when :keepalt is used.
An alternate file name is remembered for each window.

:keepa :keepalt

:keepa[lt] {cmd} Execute {cmd} while keeping the current alternate file name. Note that commands invoked indirectly (e.g., with a function) may still set the alternate file name.

All file names are remembered in the buffer list. When you enter a file name,
for editing (e.g., with ":e filename") or writing (e.g., with ":w filename"),
the file name is added to the list. You can use the buffer list to remember
which files you edited and to quickly switch from one file to another (e.g.,
to copy text) with the CTRL-^ command. First type the number of the file
and then hit CTRL-^.

CTRL-G or CTRL-G :f :fi :file

:f[ile] Prints the current file name (as typed, unless ":cd" was used), the cursor position (unless the 'ruler' option is set), and the file status (readonly, modified, read errors, new file). See the 'shortmess' option about how to make this message shorter.

:f[ile]! like :file, but don't truncate the name even when 'shortmess' indicates this.

{count}CTRL-G Like CTRL-G, but prints the current file name with full path. If the count is higher than 1 the current buffer number is also given.

g_CTRL-G word-count byte-count

g CTRL-G Prints the current position of the cursor in five ways: Column, Line, Word, Character and Byte. If the number of Characters and Bytes is the same then the Character position is omitted.

If there are characters in the line that take more than one position on the screen (<Tab> or special character), or characters using more than one byte per column (characters above 0x7F when 'encoding' is utf-8), both the byte column and the screen column are shown, separated by a dash.

Also see the 'ruler' option and the wordcount() function.

v_g_CTRL-G

{Visual}g CTRL-G Similar to "g CTRL-G", but Word, Character, Line, and Byte counts for the visually selected region are displayed.
In Blockwise mode, Column count is also shown. (For {Visual} see Visual-mode.)

:file_f

:f[ile][!] {name} Sets the current file name to {name}. The optional ! avoids truncating the message, as with :file.
If the buffer did have a name, that name becomes the alternate-file name. An unlisted buffer is created to hold the old name.

See nvim_buf_set_name() to avoid filename escaping.

:0file

:0f[ile][!] Remove the name of the current buffer. The optional ! avoids truncating the message, as with :file.

:buffers
:files
:ls List all the currently known file names. See windows.txt :files :buffers :ls.

Vim will remember the full path name of a file name that you enter. In most cases when the file name is displayed only the name you typed is shown, but the full path name is being used if you used the ":cd" command :cd.

home-replace

If the environment variable $HOME is set, and the file name starts with that string, it is often displayed with HOME replaced with "~". This was done to keep file names short. When reading or writing files the full name is still used, the "~" is only used when displaying file names. When replacing the file name would result in just "~", "~/" is used instead (to avoid confusion between options set to $HOME with 'backupext' set to "~").

When writing the buffer, the default is to use the current file name. Thus when you give the "ZZ" or ":wq" command, the original file will be overwritten. If you do not want this, the buffer can be written into another file by giving a file name argument to the ":write" command. For example:

```
vim testfile
[change the buffer with editor commands]
:w newfile
:q
```

This will create a file "newfile", that is a modified copy of "testfile".
The file "testfile" will remain unchanged. Anyway, if the 'backup' option is set, Vim renames or copies the original file before it will be overwritten. You can use this file if you discover that you need the original file. See also the 'patchmode' option. The name of the backup file is normally the same as the original file with 'backupext' appended. The default "~" is a bit strange to avoid accidentally overwriting existing files. If you prefer ".bak" change the 'backupext' option. The backup file can be placed in another directory by setting 'backupdir'.

When you started editing without giving a file name, "No File" is displayed in messages. If the ":write" command is used with a file name argument, the file name for the current file is set to that file name. This only happens when the 'F' flag is included in 'cpoptions' (by default it is included) cpo-F.
This is useful when entering text in an empty buffer and then writing it to a file. If 'cpoptions' contains the 'f' flag (by default it is NOT included) cpo-f the file name is set for the ":read file" command. This is useful when starting Vim without an argument and then doing ":read file" to start editing a file.
When the file name was set and 'filetype' is empty the filetype detection autocommands will be triggered.

not-edited

Because the file name was set without really starting to edit that file, you are protected from overwriting that file. This is done by setting the "notedited" flag. You can see if this flag is set with the CTRL-G or ":file" command. It will include "[Not edited]" when the "notedited" flag is set. When writing the buffer to the current file name (with ":w!"), the "notedited" flag is reset.

abandon

Vim remembers whether you have changed the buffer. You are protected from losing the changes you made. If you try to quit without writing, or want to start editing another file, Vim will refuse this. In order to overrule this protection, add a '!' to the command. The changes will then be lost. For example: ":q" will not work if the buffer was changed, but ":q!" will. To see whether the buffer was changed use the "CTRL-G" command. The message includes the string "[Modified]" if the buffer has been changed, or "+" if the 'm' flag is in 'shortmess'.

If you want to automatically save the changes without asking, switch on the 'autowriteall' option. 'autowrite' is the associated Vi-compatible option that does not work for all commands.

If you want to keep the changed buffer without saving it, switch on the 'hidden' option. See hidden-buffer. Some commands work like this even when 'hidden' is not set, check the help for the command.

Editing a file

edit-a-file

:e :edit reload

:e[dit][!] [++opt] [+cmd]
  Edit the current file. This is useful to re-edit the current file, when it has been changed outside of Vim.
  :edit! discard
  If [!] is given, unsaved changes in the current buffer are discarded. Without [!] the command fails if there are unsaved changes, unless 'autowriteall' is set and the file can be written.
  Also see ++opt and +cmd.

:edit_f

:e[dit][!] [++opt] [+cmd] {file}

Edit {file}.

:edit!_f

If [!] is given, unsaved changes in the current buffer are discarded. Without [!] the command fails if there are unsaved changes, unless 'hidden' is set or 'autowriteall' is set and the file can be written. Also see ++opt and +cmd.

:edit_# :e#

:e[dit] [++opt] [+cmd] #[count]
Edit the [count]th buffer (as shown by :files). This command does the same as [count] CTRL-^. But ":e #" doesn't work if the alternate buffer doesn't have a file name, while CTRL-^ still works then. Also see ++opt and +cmd.

:ene :enew

:ene[w] Edit a new, unnamed buffer. This fails when changes have been made to the current buffer, unless 'hidden' is set or 'autowriteall' is set and the file can be written.
If 'fileformats' is not empty, the first format given will be used for the new buffer. If 'fileformats' is empty, the 'fileformat' of the current buffer is used.

:ene! :enew!

:ene[w]! Edit a new, unnamed buffer. Discard any changes to the current buffer.
Set 'fileformat' like :enew.

:fin :find

:fin[d][!] [++opt] [+cmd] {file}
Find {file} in 'path' and then :edit it.
See also: 'findfunc'.

:{count}fin[d][!] [++opt] [+cmd] {file}
Just like ":find", but use the {count} match in 'path'. Thus ":2find file" will find the second "file" found in 'path'. When there are fewer matches for the file in 'path' than asked for, you get an error message.

:ex

:ex [++opt] [+cmd] [file]
Same as :edit.

:vi :visual

:vi[sual][!] [++opt] [+cmd] [file]
When used in Ex mode: Leave Ex-mode, go back to Normal mode. Otherwise same as :edit.

:vie :view

:vie[w][!] [++opt] [+cmd] file
When used in Ex mode: Leave Ex-mode, go back to Normal mode. Otherwise same as :edit, but set 'readonly' option for this buffer.

CTRL-^ CTRL-6

CTRL-^ Edit the alternate file. Mostly the alternate file is the previously edited file. This is a quick way to toggle between two files. It is equivalent to ":e #", except that it also works when there is no file name.

If the 'autowrite' or 'autowriteall' option is on and the buffer was changed, write it.
Mostly the ^ character is positioned on the 6 key, pressing CTRL and 6 then gets you what we call CTRL-^. But on some non-US keyboards CTRL-^ is produced in another way.

{count}CTRL-^ Edit [count]th file in the buffer list (equivalent to ":e #[count]"). This is a quick way to switch between files.
See CTRL-^ above for further details.

gf E446 E447

[count]gf Edit the filename at (or after) cursor.
Mnemonic: "goto file".

- The 'isfname' option decides what a filename is.

  Trailing punctuation characters ".,:;!" are ignored.
  Escaped spaces "\ " are reduced to a single space.
- The 'path' option decides where to look for the

  file, including relative directories and wildcards.
- The 'suffixesadd' option controls which suffixes

  (file extensions) are tried when building potential filepaths.
- If the search fails, it is retried after applying

'includeexpr' to the filename.

- If a [count] is given, the count'th file that is

found in the 'path' is edited.

- If the name is a URL ("type://machine/path"), it is

handled by $VIMRUNTIME/plugin/net.lua

- Environment variables are expanded. expand-env.

- On unix: "~" is expanded.

This command fails if Vim refuses to abandon the
current file.
See CTRL-W_CTRL-F to edit the file in a new window.

If you do want to edit a new file, use:

```
:e <cfile>
```

v_gf

{Visual}[count]gf Same as "gf", but the highlighted text is used as the
name of the file to edit. 'isfname' is ignored.
Leading blanks are skipped, otherwise all blanks and
special characters are included in the file name.
(For {Visual} see Visual-mode.)

gF

[count]gF Same as "gf", except if a number follows the file
name, then the cursor is positioned on that line in
the file.
The file name and the number must be separated by a
non-filename (see 'isfname') and non-numeric
character. " line " is also recognized, like it is
used in the output of :verbose command UserCmd
White space between the filename, the separator and
the number is ignored.
Examples:

eval.c:10
eval.c @ 20
eval.c (30)
eval.c 40
eval.c line 50

v_gF

{Visual}[count]gF Same as "v_gf", but if a colon and a number appear
immediately after the end of the selected text,
then the cursor is positioned on that line in the
file. Unlike gF, only a colon is recognized as
separator.

These commands are used to start editing a single file. This means that the file is read into the buffer and the current file name is set. The file that is opened depends on the current directory, see :cd.

See read-messages for an explanation of the message that is given after the file has been read.

You can use the ":e!" command if you messed up the buffer and want to start all over again. The ":e" command is only useful if you have changed the current file name.

:filename {file}

Besides the things mentioned here, more special items for where a filename is expected are mentioned at cmdline-special.

When using a command that accepts a single file name (like ":edit file") spaces in the file name are allowed, but trailing spaces are ignored. Example: The command ":e Long File Name " will edit the file "Long File Name". When using a command that accepts more than one file name (like ":next file1 file2") embedded spaces must be escaped with a backslash.

wildcard wildcards

Wildcards in {file} are expanded, but as with file completion, 'wildignore' and 'suffixes' apply. Which wildcards are supported depends on the system. These are the common ones:

```
        ?       matches one character
        *       matches anything, including nothing
        **      matches anything, including nothing, recurses into directories
        [abc]   match 'a', 'b' or 'c'
```

To avoid the special meaning of the wildcards prepend a backslash. However, on MS-Windows the backslash is a path separator and "path\[abc]" is still seen as a wildcard when "[" is in the 'isfname' option. A simple way to avoid this is to use "path\[[]abc]", this matches the file "path\[abc]".

starstar-wildcard

Expanding "**" is possible on Unix, Win32, macOS and a few other systems (but

it may depend on your 'shell' setting on Unix and macOS. It's known to work
correctly for zsh; for bash this requires at least bash version >= 4.X).
This allows searching a directory tree. This goes up to 100 directories deep.
Note there are some commands where this works slightly differently, see
file-searching.
Example:

```
:n **/*.txt
```

Finds files:

```
aaa.txt
subdir/bbb.txt
a/b/c/d/ccc.txt
```

When non-wildcard characters are used right before or after "**" these are
only matched in the top directory. They are not used for directories further
down in the tree. For example:

```
:n /usr/inc**/types.h
```

Finds files:

```
/usr/include/types.h
/usr/include/sys/types.h
/usr/inc/old/types.h
```

Note that the path with "/sys" is included because it does not need to match
"/inc". Thus it's like matching "/usr/inc*/*/*...", not
"/usr/inc*/inc*/inc*".

backtick-expansion `-expansion
On Unix and a few other systems you can also use backticks for the file name
argument, for example:

```
:next `find . -name ver\\*.c -print`
:view `ls -t *.patch  \| head -n1`
```

Vim will run the command in backticks using the 'shell' and use the standard
output as argument for the given Vim command (error messages from the shell
command will be discarded).
To see what shell command Vim is running, set the 'verbose' option to 4. When
the shell command returns a non-zero exit code, an error message will be
displayed and the Vim command will be aborted. To avoid this make the shell
always return zero like so:

```
:next `find . -name ver\\*.c -print \|\| true`
```

The backslashes before the star are required to prevent the shell from
expanding "ver*.c" prior to execution of the find program. The backslash
before the shell pipe symbol "|" prevents Vim from parsing it as command
termination.
This also works for most other systems, with the restriction that the
backticks must be around the whole item. It is not possible to have text
directly before the first or just after the last backtick.

`=

You can have the backticks expanded as a Vim expression, instead of as an
external command, by putting an equal sign right after the first backtick,
e.g.:

```
:e `=tempname()`
```

The expression can contain just about anything, thus this can also be used to
avoid the special meaning of '"', "|", '%' and '#'. However, 'wildignore'
does apply like to other wildcards.

Environment variables in the expression are expanded when evaluating the
expression, thus this works:

```
:e `=$HOME .. '/.vimrc'`
```

This uses $HOME inside a string and it will be used literally, most likely not
what you intended:

```
:e `='$HOME' .. '/.vimrc'`
```

If the expression returns a string then names are to be separated with line
breaks. When the result is a List then each item is used as a name. Line
breaks also separate names.

Note that such expressions are only supported in places where a filename is
expected as an argument to an Ex-command.

++opt [++opt]

The [++opt] argument can be used to set some options for one command, and to
specify the behavior for bad characters. The form is:

```
++{optname}
```

Or:

```
++{optname}={value}
```

Where {optname} is one of: ++ff ++enc ++bin ++nobin ++edit

ff	or	fileformat	overrides 'fileformat'
enc	or	encoding	overrides 'fileencoding'
bin	or	binary	sets 'binary'
nobin	or	nobinary	resets 'binary'
bad			specifies behavior for bad characters
edit			for :read: keeps options as if editing a file
p			for :write: creates the file's parent directory

{value} cannot contain whitespace. It can be any valid value for the options.
Examples:

```
:e ++ff=unix
```

This edits the same file again with 'fileformat' set to "unix".

```
:w ++enc=latin1 newfile
```

This writes the current buffer to "newfile" in latin1 format.

The message given when writing a file will show "[converted]" when
'fileencoding' or the value specified with ++enc differs from 'encoding'.

There may be several ++opt arguments, separated by whitespace. They must all
appear before any +cmd argument.

++p

The "++p" flag creates the parent directory of the file if it does not exist.
For example if you edit "foo/bar/file.txt", the ":write ++p" command creates
"foo/bar/" if necessary before writing the file.

```
:edit foo/bar/file.txt
:write ++p
```

If you want :write (without "++p") to always create missing parent
directories, add this autocmd to your config:

```
" Auto-create parent directories (except for URIs "://").
au BufWritePre,FileWritePre * if @% !~# '\(://\)' | call mkdir(expand('<afile>:p:h'), 'p') | endif
```

++bad

The argument of "++bad=" specifies what happens with characters that can't be
converted and illegal bytes. It can be one of three things:

++bad=X	A single-byte character that replaces each bad character.
++bad=keep	Keep bad characters without conversion. Note that this may result in illegal bytes in your text!
++bad=drop	Remove the bad characters.

The default is like "++bad=?": Replace each bad character with a question
mark. In some places an inverted question mark is used (0xBF).

Note that not all commands use the ++bad argument, even though they do not
give an error when you add it. E.g. :write.

Note that when reading, the 'fileformat' and 'fileencoding' options will be
set to the used format. When writing this doesn't happen, thus a next write
will use the old value of the option. Same for the 'binary' option.

+cmd [+cmd]

The [+cmd] argument can be used to position the cursor in the newly opened
file, or execute any other command:

+	Start at the last line.
+{num}	Start at line {num}.
+/{pat}	Start at first line containing {pat}.
+{command}	Execute {command} after opening the new file. {command} is any Ex command.

To include a white space in the {pat} or {command}, precede it with a
backslash. Double the number of backslashes.

```
:edit  +/The\ book           file
:edit  +/dir\ dirname\\      file
:edit  +set\ dir=c:\\\\temp  file
```

Note that in the last example the number of backslashes is halved twice: Once

for the "+cmd" argument and once for the ":set" command.

file-formats

The 'fileformat' option sets the <EOL> style for a file:

'fileformat'	characters	name	
"dos"	<CR><NL> or <NL>	DOS format	DOS-format
"unix"	<NL>	Unix format	Unix-format
"mac"	<CR>	Mac format	Mac-format

When reading a file, the mentioned characters are interpreted as the <EOL>.
In DOS format (default for Windows), <CR><NL> and <NL> are both interpreted as
the <EOL>. Note that when writing the file in DOS format, <CR> characters will
be added for each single <NL>. Also see file-read.

When writing a file, the mentioned characters are used for <EOL>. For DOS
format <CR><NL> is used. Also see DOS-format-write.

You can read a file in DOS format and write it in Unix format. This will
replace all <CR><NL> pairs by <NL> (assuming 'fileformats' includes "dos"):

```
:e file
:set fileformat=unix
:w
```

If you read a file in Unix format and write with DOS format, all <NL>
characters will be replaced with <CR><NL> (assuming 'fileformats' includes
"unix"):

```
:e file
:set fileformat=dos
:w
```

If you start editing a new file and the 'fileformats' option is not empty
(which is the default), Vim will try to detect whether the lines in the file
are separated by the specified formats. When set to "unix,dos", Vim will
check for lines with a single <NL> (as used on Unix) or by a <CR><NL> pair
(MS-Windows). Only when ALL lines end in <CR><NL>, 'fileformat' is
set to "dos", otherwise it is set to "unix". When 'fileformats' includes
"mac", and no <NL> characters are found in the file, 'fileformat' is set to
"mac".

If the 'fileformat' option is set to "dos" on non-MS-Windows systems the
message "[dos]" is shown to remind you that something unusual is happening. On
MS-Windows systems you get the message "[unix]" if 'fileformat' is set to
"unix". On all systems you get the message "[mac]" if 'fileformat' is set to
"mac".

If the 'fileformats' option is empty and DOS format is used, but while reading
a file some lines did not end in <CR><NL>, "[CR missing]" will be included in
the file message.
If the 'fileformats' option is empty and Mac format is used, but while reading
a file a <NL> was found, "[NL missing]" will be included in the file message.

If the new file does not exist, the 'fileformat' of the current buffer is used
when 'fileformats' is empty. Otherwise the first format from 'fileformats' is
used for the new file.

Before editing binary, executable or Vim script files you should set the
'binary' option. A simple way to do this is by starting Vim with the "-b"
option. This will avoid the use of 'fileformat'. Without this you risk that
single <NL> characters are unexpectedly replaced with <CR><NL>.

END-OF-LINE AND END-OF-FILE **eol-and-eof**

Vim has several options to control the file format:

'fileformat'	the <EOL> style: Unix, DOS, Mac
'endofline'	whether the last line ends with a <EOL>
'endoffile'	whether the file ends with a CTRL-Z
'fixendofline'	whether to fix eol and eof

The first three values are normally detected automatically when reading the
file and are used when writing the text to a file. While editing the buffer
it looks like every line has a line ending and the CTRL-Z isn't there (an
exception is when 'binary' is set, it works differently then).

The 'fixendofline' option can be used to choose what to write. You can also
change the option values to write the file differently than how it was read.

Here are some examples how to use them.

If you want files in Unix format (every line NL terminated):

```
setl ff=unix fixeol
```

You should probably do this on any Unix-like system. Also modern MS-Windows
systems tend to work well with this. It is recommended to always use this
format for Vim scripts.

If you want to use an old MS-DOS file in a modern environment, fixing line
endings and dropping CTRL-Z, but keeping the <CR><NL> style <EOL>:

```
setl ff=dos fixeol
```

This is useful for many MS-Windows programs, they regularly expect the
<CR><NL> line endings.

If you want to drop the final <EOL> and add a final CTRL-Z (e.g. for an old
system like CP/M):

```
setl ff=dos nofixeol noeol eof
```

If you want to preserve the fileformat exactly as-is, including any final
<EOL> and final CTRL-Z:

```
setl nofixeol
```

OPENING FILES PROGRAMMATICALLY **buffer-open** **open-file**

When a plugin already has a literal filepath (e.g. from a picker, LSP result,
or filesystem scan) and just wants to open it, :edit is the wrong tool: its
argument is subject to cmdline-special expansion, wildcard expansion, etc.
fnameescape() escapes only some of these, and may depend on 'isfname'.

Instead, to open a filepath literally, use bufadd(vim.fs.normalize(…)):

```
local buf = vim.fn.bufadd(vim.fs.normalize(path))
-- Set 'buflisted'; trigger BufReadPre/BufReadPost/FileType.
vim.api.nvim_buf_call(buf, vim.cmd.edit)
```

Alternatively, nvim_cmd() and vim.cmd() accept a structured form whose
args are tokenized verbatim. But the argument still goes through :edit's
filename-expansion phase, so some parts of path are still expanded:

```
vim.cmd{ cmd = 'edit',
  args = { path },
  magic = { file = false, bar = false },
}
```

The argument list **argument-list** **arglist**

If you give more than one file name when starting Vim, this list is remembered
as the argument list. You can jump to each file in this list.

Do not confuse this with the buffer list, which you can see with the
:buffers command. The argument list was already present in Vi, the buffer
list is new in Vim. Every file name in the argument list will also be present
in the buffer list (unless it was deleted with :bdel or :bwipe). But it's
common that names in the buffer list are not in the argument list.

This subject is introduced in section 07.2 of the user manual.

There is one global argument list, which is used for all windows by default.
It is possible to create a new argument list local to a window, see
:arglocal.

You can use the argument list with the following commands, and with the
expression functions argc() and argv(). These all work on the argument
list of the current window.

:ar[gs] :ar :arg :args
Print the argument list, with the current file in
square brackets.

:ar[gs] [++opt] [+cmd] {arglist} :args_f

Define {arglist} as the new argument list and edit
the first one. This fails when changes have been made
and Vim does not want to abandon the current buffer.
Also see ++opt and +cmd.

:ar[gs]! [++opt] [+cmd] {arglist} :args_f!

Define {arglist} as the new argument list and edit
the first one. Discard any changes to the current
buffer.
Also see ++opt and +cmd.

:[count]arge[dit][!] [++opt] [+cmd] {name} ... :arge :argedit
Add {name}s to the argument list and edit it.
There is no check for duplicates, it is possible to
add a file to the argument list twice :argded.
This is like using :argadd and then :edit (with
the small exception that :edit does not change the
argument list, so the argument list pointer isn't
changed).
Spaces in filenames have to be escaped with "\".
[count] is used like with :argadd.
If the current file cannot be abandoned {name}s will
still be added to the argument list, but won't be
edited. No check for duplicates is done.
Also see ++opt and +cmd.

:[count]arga[dd] {name} ... :arga :argadd E479
:[count]arga[dd] E1156

Add the {name}s to the argument list. When {name} is omitted add the current buffer name to the argument list.
If [count] is omitted, the {name}s are added just after the current entry in the argument list.
Otherwise they are added after the [count]'th file.
If the argument list is "a b c", and "b" is the current argument, then these commands result in:

command	new argument list
:argadd x	a [b] x c
:0argadd x	x a [b] c
:1argadd x	a x [b] c
:$argadd x	a [b] c x

And after the last one:

:+2argadd y	a [b] c x y

There is no check for duplicates, it is possible to add a file to the argument list twice. You can use :argdedupe to fix it afterwards:

```
:argadd *.txt | argdedupe
```

The currently edited file is not changed.
Note: you can also use this method:

```
:args ## x
```

This will add the "x" item and sort the new list.

:argded[upe] :argded :argdedupe
Remove duplicate filenames from the argument list.
If your current file is a duplicate, your current file will change to the original file index.

:argd[elete] {pattern} ... :argd :argdelete E480 E610
Delete files from the argument list that match the {pattern}s. {pattern} is used like a file pattern, see file-pattern. "%" can be used to delete the current entry.
This command keeps the currently edited file, also when it's deleted from the argument list.
Example:

```
:argdel *.obj
```

:[range]argd[elete] Delete the [range] files from the argument list.
Example:

```
:10,$argdel
```

Deletes arguments 10 and further, keeping 1-9.

```
:$argd
```

Deletes just the last one.

```
:argd
:.argd
```

Deletes the current argument.

```
:%argd
```

Removes all the files from the arglist.
When the last number in the range is too high, up to the last argument is deleted.

:argu :argument
:[count]argu[ment] [count] [++opt] [+cmd]
Edit file [count] in the argument list. When [count] is omitted the current entry is used. This fails when changes have been made and Vim does not want to abandon the current buffer.
Also see ++opt and +cmd.

:[count]argu[ment]! [count] [++opt] [+cmd]
Edit file [count] in the argument list, discard any changes to the current buffer. When [count] is omitted the current entry is used.
Also see ++opt and +cmd.

:[count]n[ext] [++opt] [+cmd] :n :ne :next E165 E163
Edit [count] next file. This fails when changes have been made and Vim does not want to abandon the current buffer. Also see ++opt and +cmd.

]a

]a Mapped to :next. default-mappings

:[count]n[ext]! [++opt] [+cmd]
Edit [count] next file, discard any changes to the buffer. Also see ++opt and +cmd.

:n[ext] [++opt] [+cmd] {arglist} :next_f

Same as :args_f.

:n[ext]! [++opt] [+cmd] {arglist}
Same as :args_f!.

:[count]N[ext] [count] [++opt] [+cmd] :Next :N E164
Edit [count] previous file in argument list. This fails when changes have been made and Vim does not want to abandon the current buffer.
Also see ++opt and +cmd.

:[count]N[ext]! [count] [++opt] [+cmd]
Edit [count] previous file in argument list. Discard any changes to the buffer. Also see ++opt and +cmd.

:[count]prev[ious] [count] [++opt] [+cmd] :prev :previous
Same as :Next. Also see ++opt and +cmd.

[a

[a Mapped to :previous. default-mappings

:rew :rewind
:rew[ind] [++opt] [+cmd]
Start editing the first file in the argument list. This fails when changes have been made and Vim does not want to abandon the current buffer.
Also see ++opt and +cmd.

[A

[A Mapped to :rewind. default-mappings

:rew[ind]! [++opt] [+cmd]
Start editing the first file in the argument list. Discard any changes to the buffer. Also see ++opt and +cmd.

:fir :first
:fir[st][!] [++opt] [+cmd]
Other name for ":rewind".

:la :last
:la[st] [++opt] [+cmd]
Start editing the last file in the argument list. This fails when changes have been made and Vim does not want to abandon the current buffer.
Also see ++opt and +cmd.

]A

]A Mapped to :last. default-mappings

:la[st]! [++opt] [+cmd]
Start editing the last file in the argument list. Discard any changes to the buffer. Also see ++opt and +cmd.

:wn :wnext
:[count]wn[ext] [++opt]
Write current file and start editing the [count] next file. Also see ++opt and +cmd.

:[count]wn[ext] [++opt] {file}
Write current file to {file} and start editing the [count] next file, unless {file} already exists and the 'writeany' option is off. Also see ++opt and +cmd.

:[count]wn[ext]! [++opt] {file}
Write current file to {file} and start editing the [count] next file. Also see ++opt and +cmd.

:[count]wN[ext][!] [++opt] [file] :wN :wNext
:[count]wp[revious][!] [++opt] [file] :wp :wprevious
Same as :wnext, but go to previous file instead of next.

The [count] in the commands above defaults to one. For some commands it is possible to use two counts. The last one (rightmost one) is used.

If no [+cmd] argument is present, the cursor is positioned at the last known cursor position for the file. If 'startofline' is set, the cursor will be positioned at the first non-blank in the line, otherwise the last know column is used. If there is no last known cursor position the cursor will be in the first line (the last line in Ex mode).

{arglist}

The wildcards in the argument list are expanded and the file names are sorted. Thus you can use the command vim *.c to edit all the C files. From within Vim the command :n *.c does the same.

White space is used to separate file names. Put a backslash before a space or tab to include it in a file name. E.g., to edit the single file "foo bar":

```
:next foo\ bar
```

On Unix and a few other systems you can also use backticks, for example:

```
:next `find . -name \\*.c -print`
```

The backslashes before the star are required to prevent "*.c" to be expanded
by the shell before executing the find program.

arglist-position

When there is an argument list you can see which file you are editing in the
title of the window (if there is one and 'title' is on) and with the file
message you get with the "CTRL-G" command. You will see something like
        (4 of 11)
If you are not really editing the file at the current position in the argument
list it will be
        ((4) of 11)
This means that you are position 4 in the argument list, but not editing the
fourth file in the argument list. This happens when you do ":e file".

LOCAL ARGUMENT LIST

:arglocal

:argl[ocal]   Make a local copy of the global argument list.
                        Doesn't start editing another file.

:argl[ocal][!] [++opt] [+cmd] {arglist}
                        Define a new argument list, which is local to the
                        current window. Works like :args_f otherwise.

:argglobal

:argg[lobal]   Use the global argument list for the current window.
                        Doesn't start editing another file.

:argg[lobal][!] [++opt] [+cmd] {arglist}
                        Use the global argument list for the current window.
                        Define a new global argument list like :args_f.
                        All windows using the global argument list will see
                        this new list.

There can be several argument lists. They can be shared between windows.
When they are shared, changing the argument list in one window will also
change it in the other window.

When a window is split the new window inherits the argument list from the
current window. The two windows then share this list, until one of them uses
:arglocal or :argglobal to use another argument list.

USING THE ARGUMENT LIST

:argdo

:[range]argdo[!] {cmd} Execute {cmd} for each file in the argument list or,
                        if [range] is specified, only for arguments in that
                        range. It works like doing this:

```
:rewind
:{cmd}
:next
:{cmd}
etc.
```

                        When the current file can't be abandoned and the [!]
                        is not present, the command fails.
                        When an error is detected on one file, further files
                        in the argument list will not be visited.
                        The last file in the argument list (or where an error
                        occurred) becomes the current file.
                        {cmd} can contain '|' to concatenate several commands.
                        {cmd} must not change the argument list.
                        Note: While this command is executing, the Syntax
                        autocommand event is disabled by adding it to
                        'eventignore'. This considerably speeds up editing
                        each file.
                        Also see :windo, :tabdo, :bufdo, :cdo, :ldo,
                        :cfdo and :lfdo.

Example:

```
:args *.c
:argdo set ff=unix | update
```

This sets the 'fileformat' option to "unix" and writes the file if it is now
changed. This is done for all *.c files.

Example:

```
:args *.[ch]
:argdo %s/\<my_foo\>/My_Foo/ge | update
```

This changes the word "my_foo" to "My_Foo" in all "*.c" and "*.h" files. The "e"
flag is used for the ":substitute" command to avoid an error for files where
"my_foo" isn't used. ":update" writes the file only if changes were made.

Writing

writing save-file

Note: When the 'write' option is off, you are not able to write any file.

:w :write
E502 E503 E504 E505
E512 E514 E667 E949

:w[rite] [++opt]
Write the whole buffer to the current file. This is the normal way to save changes to a file. Fails when 'readonly' is set or when there is another reason why the file can't be written, such as when the parent directory doesn't exist (use ++p to avoid that). For ++opt see ++opt, but only ++p, ++bin, ++nobin, ++ff and ++enc are effective.

:w[rite]! [++opt]
Like ":write", but forcefully write when 'readonly' is set or there is another reason why writing was refused.
Note: This may change the permission and ownership of the file and break (symbolic) links. Add the 'W' flag to 'cpoptions' to avoid this.

:[range]w[rite][!] [++opt]
Write the specified lines to the current file. This is unusual, because the file will not contain all lines in the buffer.

:w_f :write_f
:[range]w[rite] [++opt] {file}
Write the specified lines to {file}, unless it already exists and the 'writeany' option is off.

:w!

:[range]w[rite]! [++opt] {file}
Write the specified lines to {file}. Overwrite an existing file.

:w a :write a E494
:[range]w[rite][!] [++opt] >

```
Append the specified lines to the current file.
```

:[range]w[rite][!] [++opt] >> {file}
Append the specified lines to {file}. '!' forces the write even if file does not exist.

:w_c :write_c
:[range]w[rite] [++opt] !{cmd}
Execute {cmd} with [range] lines as standard input (note the space in front of the '!'). {cmd} is executed like with ":!{cmd}", any '!' is replaced with the previous command :!.

The default [range] for the ":w" command is the whole buffer (1,$). The '[and '] marks will be set to the [range] being used for the write command.
If you write the whole buffer, it is no longer considered changed. When you write it to a different file with ":w somefile" it depends on the "+" flag in 'cpoptions'. When included, the write command will reset the 'modified' flag, even though the buffer itself may still be different from its file.

If a file name is given with ":w" it becomes the alternate file. This can be used, for example, when the write fails and you want to try again later with ":w #". This can be switched off by removing the 'A' flag from the 'cpoptions' option.

Note that the 'fsync' option matters here. If it's set it may make writes slower (but safer).

:sav :saveas
:sav[eas][!] [++opt] {file}
Save the current buffer under the name {file} and set the filename of the current buffer to {file}. The previous name is used for the alternate file name.
The [!] is needed to overwrite an existing file.
When 'filetype' is empty filetype detection is done with the new name, before the file is written.
When the write was successful 'readonly' is reset.

:up :update
:[range]up[date][!] [++opt] [>>] [file]
Like ":write", but only write when the buffer has been modified, or when the buffer represents a new file that doesn't exist on disk.

WRITING WITH MULTIPLE BUFFERS

buffer-write

:wa :wall

:wa[ll] [++opt] Write all changed buffers. Buffers without a file name cause an error message. Buffers which are readonly are not written.
For ++opt see ++opt, but only ++p is effective, and applies to each written file.

:wa[ll]! [++opt] Write all changed buffers, even the ones that are readonly. Buffers without a file name are not written and cause an error message.

Vim will warn you if you try to overwrite a file that has been changed elsewhere (unless "!" was used). See timestamp.

backup E207 E506 E507 E508 E509 E510

If you write to an existing file (but do not append) while the 'backup', 'writebackup' or 'patchmode' option is on, a backup of the original file is made. The file is either copied or renamed (see 'backupcopy'). After the file has been successfully written and when the 'writebackup' option is on and the 'backup' option is off, the backup file is deleted. When the 'patchmode' option is on the backup file may be renamed.

backup-table

'backup'	'writebackup'	action
off	off	no backup made
off	on	backup current file, deleted afterwards (default)
on	off	delete old backup, backup current file
on	on	delete old backup, backup current file

When the 'backupskip' pattern matches with the name of the file which is written, no backup file is made. The values of 'backup' and 'writebackup' are ignored then.

When the 'backup' option is on, an old backup file (with the same name as the new backup file) will be deleted. If 'backup' is not set, but 'writebackup' is set, an existing backup file will not be deleted. The backup file that is made while the file is being written will have a different name.

On some filesystems it's possible that in a crash you lose both the backup and the newly written file (it might be there but contain bogus data). In that case try recovery, because the swap file is synced to disk and might still be there. :recover

The directories given with the 'backupdir' option are used to put the backup file in. (default: same directory as the written file).

Whether the backup is a new file, which is a copy of the original file, or the original file renamed depends on the 'backupcopy' option. See there for an explanation of when the copy is made and when the file is renamed.

If the creation of a backup file fails, the write is not done. If you want to write anyway add a '!' to the command.

file-watcher

When you notice issues with programs, that act upon when a buffer is written (like inotify, entr or fswatch) or when external applications execute Vim to edit the file (like git) and those programs do not seem to notice that the original file has been changed, you may want to consider switching the 'backupcopy' option value to "yes". This makes sure, Vim writes to the same file, that those watcher programs expect, without creating a new file (which prevents them from detecting that the file has changed). See also crontab

write-permissions

When writing a new file the permissions are read-write. For unix the mask is 0o666 with additionally umask applied. When writing a file that was read Vim will preserve the permissions, but clear the s-bit.

write-readonly

When the 'cpoptions' option contains 'W', Vim will refuse to overwrite a readonly file. When 'W' is not present, ":w!" will overwrite a readonly file, if the system allows it (the directory must be writable).

write-fail

If the writing of the new file fails, you have to be careful not to lose your changes AND the original file. If there is no backup file and writing the new file failed, you have already lost the original file! DON'T EXIT VIM UNTIL YOU WRITE OUT THE FILE! If a backup was made, it is put back in place of the original file (if possible). If you exit Vim, and lose the changes you made, the original file will mostly still be there. If putting back the original file fails, there will be an error message telling you that you lost the original file.

DOS-format-write

If the 'fileformat' is "dos", <CR><NL> is used for <EOL>. This is default for Windows. On other systems the message "[dos]" is shown to remind you that an unusual <EOL> was used.

Unix-format-write

If the 'fileformat' is "unix", <NL> is used for <EOL>. On Windows the message "[unix]" is shown.

Mac-format-write

If the 'fileformat' is "mac", <CR> is used for <EOL>. The
message "[mac]" is shown.

See also file-formats and the 'fileformat' and 'fileformats' options.

ACL

ACL stands for Access Control List. It is an advanced way to control access
rights for a file. It is used on new MS-Windows and Unix systems, but only
when the filesystem supports it.
   Vim attempts to preserve the ACL info when writing a file. The backup file
will get the ACL info of the original file.
   The ACL info is also used to check if a file is read-only (when opening the
file).

xattr E1506 E1508 E1509

xattr stands for Extended Attributes. It is an advanced way to save metadata
alongside the file in the filesystem. It depends on the actual filesystem
being used and Vim supports it only on a Linux system.
   Vim attempts to preserve the extended attribute info when writing a file.
The backup file will get the extended attribute of the original file.

read-only-share

When MS-Windows shares a drive on the network it can be marked as read-only.
This means that even if the file read-only attribute is absent, and the ACL
settings on NT network shared drives allow writing to the file, you can still
not write to the file. Vim on Win32 platforms will detect read-only network
drives and will mark the file as read-only. You will not be able to override
it with :write.

write-device

When the file name is actually a device name, Vim will not make a backup (that
would be impossible). You need to use "!", since the device already exists.
Example for Unix:

```
:w! /dev/lpt0
```

and MS-Windows:

```
:w! lpt0
```

For Unix a device is detected when the name doesn't refer to a normal file or
a directory. A fifo or named pipe also looks like a device to Vim.
For MS-Windows the device is detected by its name:
        CON
        CLOCK$
        NUL
        PRN
        COMn   n=1,2,3... etc
        LPTn   n=1,2,3... etc
The names can be in upper- or lowercase.

Writing and quitting

write-quit

:q :quit

:q[uit] Quit the current window. Quit Vim if this is the last edit-window. This fails when changes have been made and Vim refuses to abandon the current buffer, and when the last file in the argument list has not been edited.
If there are other tabpages and quitting the last window in the current tabpage, the current tabpage is closed.
Triggers the QuitPre autocommand event.
See CTRL-W_q for quitting another window.

:conf[irm] q[uit] Quit, but give prompt when changes have been made, or the last file in the argument list has not been edited. See :confirm and 'confirm'.

:q[uit]! Quit without writing, also when the current buffer has changes. The buffer is unloaded, also when it has 'hidden' set.
If this is the last window and there is a modified hidden buffer, the current buffer is abandoned and the first changed hidden buffer becomes the current buffer.
Use ":qall!" to exit always.

:cq[uit] Quit always, without writing, and return an error code. See :cq.

:wq

:wq [++opt] Write the current file and close the window. If this was the last edit-window Vim quits.
Writing fails when the file is read-only or the buffer does not have a name. Quitting fails when the last file in the argument list has not been edited.

:wq! [++opt] Write the current file and close the window. If this was the last edit-window Vim quits. Writing fails when the current buffer does not have a name.

:wq [++opt] {file} Write to {file} and close the window. If this was the last edit-window Vim quits. Quitting fails when the last file in the argument list has not been edited.

:wq! [++opt] {file} Write to {file} and close the current window. Quit Vim if this was the last edit-window.

:[range]wq[!] [++opt] [file]
Same as above, but only write the lines in [range].

:x :xit
:[range]x[it][!] [++opt] [file]
Like ":wq", but write only when changes have been made.
When 'hidden' is set and there are more windows, the current buffer becomes hidden, after writing the file.

:exi :exit
:[range]exi[t][!] [++opt] [file]
Same as :xit.

ZZ

ZZ Write current file, if modified, and close the current window (same as ":x").
If there are several windows for the current file, only the current window is closed.

ZQ

ZQ Quit without checking for changes (same as ":q!").

ZR

[count]ZR Performs :restart. If a [count] of [1-8] is given, the session is not restored (":restart! +qall"). If a [count] of 9 is given, restarts without restoring the session or checking for changes (":restart! +qall!").

MULTIPLE WINDOWS AND BUFFERS **window-exit**

:qa :qall
:qa[ll] Exit Vim, unless there are some buffers which have been changed. (Use ":bmod" to go to the next modified buffer).
When 'autowriteall' is set all changed buffers will be written, like :wqall.

:conf[irm] qa[ll]
Exit Vim. Bring up a prompt when some buffers have been changed. See :confirm.

:qa[ll]! Exit Vim. Any changes to buffers are lost.
Also see :cquit, it does the same but exits with a non-zero value.

:quita :quitall
:quita[ll][!] Same as ":qall".

:wqa[ll] [++opt] :wqa :wqall :xa :xall
:xa[ll] Write all changed buffers and exit Vim. If there are buffers without a file name, which are readonly or which cannot be written for another reason, Vim will not quit.

:conf[irm] wqa[ll] [++opt]
:conf[irm] xa[ll]
Write all changed buffers and exit Vim. Bring up a prompt when some buffers are readonly or cannot be written for another reason. See :confirm.

:wqa[ll]! [++opt]
:xa[ll]! Write all changed buffers, even the ones that are readonly, and exit Vim. If there are buffers without a file name or which cannot be written for another reason, Vim will not quit.

Dialogs **edit-dialogs**

:confirm :conf
:conf[irm] {command} Execute {command}, and use a dialog when an operation has to be confirmed. Can be used on the :edit, :restart, :q, :qa and :w commands (the latter to override a read-only setting), and any commands that can fail because of unsaved changes, such as :only, :buffer, :bdelete, etc.

Examples:

```
:confirm w foo
```

Will ask for confirmation when "foo" already exists.

```
:confirm q
```

Will ask for confirmation when there are changes.

```
:confirm qa
```

If any modified, unsaved buffers exist, you will be prompted to save or abandon each one. There are also choices to "save all" or "abandon all".

If you want to always use ":confirm", set the 'confirm' option.

:browse :bro

:bro[wse] {command} [file]
Without [file]: opens the current-directory (with the dir browser by default; see dir-disable).

{command} must be one of: :edit, :split, :vsplit, :tabedit, :tabnew.

With a [file] argument, or if {command} does not support browsing: executes {command}.

Examples:

```
:browse edit
```

Open the current directory.

```
:browse vsplit
```

Open the current directory in a vertical split.

See :oldfiles for ":browse oldfiles".

browsefilter

For MS-Windows you can modify the filters that are used in the browse dialog. By setting the g:browsefilter or b:browsefilter variables, you can change the filters globally or locally to the buffer. The variable is set to a string in the format "{filter label}\t{pattern};{pattern}\n" where "{filter label}" is the text that appears in the "Files of Type" comboBox, and {pattern} is the pattern which filters the filenames. Several patterns can be given, separated by ';'.

For example, to have only Vim files in the dialog, you could use the following command:

```
let g:browsefilter = "Vim scripts\t*.vim\nVim Startup Files\t*vimrc\n"
```

You can override the filter setting on a per-buffer basis by setting the b:browsefilter variable. You would most likely set b:browsefilter in a filetype plugin, so that the browse dialog would contain entries related to the type of file you are currently editing. Disadvantage: This makes it difficult to start editing a file of a different type. To overcome this, you can add the following as the final filter on Windows:

```
All Files\t(*.*)\t*\n
```

Or the following on other platforms, so that the user can still access any desired file:

```
All Files\t(*)\t*\n
```

To avoid setting browsefilter when Vim does not actually support it, you can use has("browsefilter"):

```
if has("browsefilter")
   let g:browsefilter = "whatever"
endif
```

The current directory

current-directory

The "effective working directory" affects anything that operates on relative paths, e.g. ":edit ./foo.txt" or shell commands like ":!ls" or ":te ls". There are four current-directory "scopes": global, tab, window and buffer. You can use :cd, :tcd, :lcd and :bcd to change to a directory at a given scope.

The precedence of scopes is ordered from narrowest to widest:

```
buffer > window > tab > global
```

Use getcwd() to get the effective directory, and :verbose pwd to also see which scope it came from.

Setting a directory at a higher scope clears the lower ones: :tcd clears the window-local directory and :cd clears both. But the buffer-local directory is never cleared, except by :bcd or :bcd!.

:chd[ir] :cd :chd :chdir E747 E472
:cd Change the current directory to the home directory, unsetting the same directories as ":cd {path}".

If 'cdhome' is disabled this only prints the name. Use :pwd to print it regardless of 'cdhome'.

:cd {path}

Change the current directory to {path}, resolved against 'cdpath' (if {path} is relative). On MS-Windows this also changes the active drive.

Clears the window-local and tab-local directories (on the current window/tab), but not the buffer-local one.

To change to the directory of the current file:

```
:cd %:h
```

Does not change the meaning of an already opened file, because its full path name is remembered. Files from the arglist may change though!

:cd- E186

:cd -

Change to previous current-directory (before the last ":cd {path}" command).

:tch[dir] :tc :tcd :tch :tchdir

:tc[d] [path]

Like :cd, but only set the directory for the current tab. Clears the window-local directory of the current window, but not the buffer-local one.

Does not change the current directory for other tabs, nor any other window or buffer with its own local directory.

"Sticky" behavior: a new tab inherits the current tab-local directory.

:tcd-

:tc[d] -

Change to previous tab-local directory (before the last ":tcd {path}" command).

:lch[dir] :lc :lcd :lch :lchdir

:lc[d] [path]

Like :cd, but only set the directory for the current window. Does not change the current directory for other tabs, windows or buffers.

The specified directory becomes the current directory whenever that window is entered, unless its current buffer has a buffer-local (:bcd) directory.

"Sticky" behavior: a new window inherits the current window-local directory.

:lcd-

:lc[d] -

Change to previous window-local directory (before the last ":lcd {path}" command).

:bch[dir] :bc :bcd :bch :bchdir

:bc[d] [path]

Like :cd, but only set the directory for the current buffer. Does not change the current directory for tabs, windows or other buffers.

Not "sticky": unlike :lcd and :tcd, a new window or buffer (:edit, :enew, …) will not "inherit" the current buffer-local directory. Use :tcd or :lcd if you want "sticky" behavior.

See also project-dir.

:bcd-

:bc[d] -

Change to previous buffer-local directory (before the last ":bcd {path}" command).

:lcd! :tcd! :bcd!

:lc[d]!

Unset the window-local current-directory.

:tc[d]!

Unset the tab-local current-directory.

:bc[d]!

Unset the buffer-local current-directory.

Each unsets only its own scope, so the next-higher scope takes effect. No-op if the scope has no local directory.

The global directory always exists, so ":cd!" is the same as :cd.

:pw :pwd E187

:pw[d]

Print the current directory name.
See also getcwd().

:pwd-verbose

When 'verbose' is non-zero, :pwd will also display what scope the current directory was set. Example:

```
" Set by :cd
:verbose pwd
[global] /path/to/current

" Set by :tcd
:verbose pwd
[tabpage] /path/to/current
```

Project directory

project-dir workspace-dir

You can use :bcd to assign a "workspace" (or "project directory") to each buffer, so commands like :make, :grep and :terminal always run relative to the buffer's "workspace".

Example: change to the project root (found by vim.fs.root()) of each buffer:

```
vim.api.nvim_create_autocmd('BufReadPost', {
  callback = function(ev)
    local root = vim.fs.root(ev.buf, { '.git', 'Makefile' })
    if root then
      vim.cmd.bcd(root)
    end
  end,
})
```

See also:
- lsp-buf-working-dir

- terminal-osc7

Editing binary files

edit-binary

Although Vim was made to edit text files, it is possible to edit binary files. The -b Vim argument (b for binary) makes Vim do file I/O in binary mode, and sets some options for editing binary files ('binary' on, 'textwidth' to 0, 'modeline' off, 'expandtab' off). Setting the 'binary' option has the same effect. Don't forget to do this before reading the file.

There are a few things to remember when editing binary files:
- When editing executable files the number of bytes must not change.

  Use only the "R" or "r" command to change text. Do not delete characters with "x" or by backspacing.
- Set the 'textwidth' option to 0. Otherwise lines will unexpectedly be

  split in two.
- When there are not many <EOL>s, the lines will become very long. If you

  want to edit a line that does not fit on the screen reset the 'wrap' option. Horizontal scrolling is used then. If a line becomes too long (see limits) you cannot edit that line. The line will be split when reading the file. It is also possible that you get an "out of memory" error when reading the file.
- Make sure the 'binary' option is set BEFORE loading the

  file. Otherwise both <CR><NL> and <NL> are considered to end a line and when the file is written the <NL> will be replaced with <CR><NL>.
- <Nul> characters are shown on the screen as ^@. You can enter them with

  "CTRL-V CTRL-@" or "CTRL-V 000".
- To insert a <NL> character in the file split a line. When writing the

  buffer to a file a <NL> will be written for the <EOL>.
- Vim normally appends an <EOL> at the end of the file if there is none.

  Setting the 'binary' option prevents this. If you want to add the final <EOL>, set the 'endofline' option. You can also read the value of this option to see if there was an <EOL> for the last line (you cannot see this in the text).

Encryption

encryption

:X E817 E818 E819 E820

Support for editing encrypted files has been removed.
https://github.com/neovim/neovim/issues/694
https://github.com/neovim/neovim/issues/701

Timestamps

timestamp timestamps

Vim remembers the modification timestamp, mode and size of a file when you begin editing it. This is used to avoid that you have two different versions of the same file (without you knowing this).

After a shell command (:!cmd suspend :read! K) or :terminal job exits: timestamps, file modes and file sizes are compared for all buffers in a window. Nvim will run FileChangedShell autocommands or display a warning for any files that changed.

E321 E462

If you want to automatically reload a file when it has been changed outside of Vim, set the 'autoread' option. This doesn't work at the moment you write the file though, only when the file wasn't changed inside of Vim.

Nvim uses file system watchers to detect external changes in real-time for all loaded buffers when 'autoread' is set. This means files are reloaded promptly without waiting for FocusGained or :checktime. The watchers are managed automatically: started when a buffer is loaded, restarted after :write, and stopped when the buffer is deleted or 'autoread' is disabled.

ignore-timestamp

If you do not want to be asked or automatically reload the file, you can use this:

```
set buftype=nofile
```

Or, when starting gvim from a shell:

```
gvim file.log -c "set buftype=nofile"
```

Note that if a FileChangedShell autocommand is defined you will not get a warning message or prompt. The autocommand is expected to handle this.

There is no warning for a directory. But you do get warned if you started editing a new file and it was created as a directory later.

When Vim notices the timestamp of a file has changed, and the file is being edited in a buffer but has not changed, Vim checks if the contents of the file is equal. This is done by reading the file again (into a hidden buffer, which is immediately deleted again) and comparing the text. If the text is equal, you will get no warning.

If you don't get warned often enough you can use the following command.

:checkt :checktime

:checkt[ime] Check if any buffers were changed outside of Vim.
Check and warn you if you would end up with two versions of a file.
If this is called from an autocommand, a ":global" command or is not typed the actual check is postponed until a moment the side effects (reloading the file) would be harmless.
Each loaded buffer is checked for its associated file being changed. If the file was changed Vim will take action. If there are no changes in the buffer and 'autoread' is set, the buffer is reloaded. Otherwise, you are offered the choice of reloading the file. If the file was deleted you get an error message.
If the file previously didn't exist you get a warning if it exists now.
Once a file has been checked the timestamp is reset, you will not be warned again.
Syntax highlighting, marks, diff status, 'fileencoding', 'fileformat' and 'binary' options are not changed. See v:fcs_choice to reload these too (for example, if a code formatting tools has changed the file).

:[N]checkt[ime] {filename}
:[N]checkt[ime] [N]
Check the timestamp of a specific buffer. The buffer may be specified by name, number or with a pattern.

E813 E814

Vim will reload the buffer if you chose to. If a window is visible that contains this buffer, the reloading will happen in the context of this window. Otherwise a special window is used, so that most autocommands will work. You can't close this window. A few other restrictions apply. Best is to make sure nothing happens outside of the current buffer. E.g., setting window-local options may end up in the wrong window. Splitting the window, doing something there and closing it should be OK (if there are no side effects from other autocommands). Closing unrelated windows and buffers will get you into trouble.

Before writing a file, the timestamp is checked (unless "!" was used). If it has changed, Vim will ask if you really want to overwrite the file:

> **WARNING:** The file has been changed since reading it!!!
> Do you really want to write to it (y/n)?

If you hit 'y' Vim will continue writing the file. If you hit 'n' the write is aborted. If you used ":wq" or "ZZ" Vim will not exit, you will get another chance to write the file.

The message would normally mean that somebody has written to the file after the edit session started. This could be another person, in which case you probably want to check if your changes to the file and the changes from the other person should be merged. Write the file under another name and check for differences (the "diff" program can be used for this).

It is also possible that you modified the file yourself, from another edit session or with another command (e.g., a filter command). Then you will know which version of the file you want to keep.

The accuracy of the time check depends on the filesystem. On Unix it is usually sub-second. With old file systems and on MS-Windows it is normally one second. Use has('nanotime') to check if sub-second time stamp checks are available.

There is one situation where you get the message while there is nothing wrong: On a Win32 system on the day daylight saving time starts. There is something in the Win32 libraries that confuses Vim about the hour time difference. The problem goes away the next day.

File Searching

file-searching

The file searching is currently used for the 'path', 'cdpath' and 'tags' options, for finddir() and findfile(). Other commands use wildcards which is slightly different.

There are three different types of searching:

1) Downward search:

starstar

Downward search uses the wildcards "*", "**" and possibly others supported by your operating system. "*" and "**" are handled inside Vim, so they work on all operating systems. Note that "**" only acts as a special wildcard when it is at the start of a name.

The usage of "*" is quite simple: It matches 0 or more characters. In a search pattern this would be .*. Note that the "." is not used for file searching.

"**" is more sophisticated:

- It ONLY matches directories.
- It matches up to 30 directories deep by default, so you can use it to search an entire directory tree
- The maximum number of levels matched can be given by appending a number to "**".
  Thus '/usr/**2' can match:

```
/usr
/usr/include
/usr/include/sys
/usr/include/g++
/usr/lib
/usr/lib/X11
....
```

  It does NOT match '/usr/include/g++/std' as this would be three levels.
  The allowed number range is 0 ("**0" is removed) to 100
  If the given number is smaller than 0 it defaults to 30, if it's bigger than 100 then 100 is used. The system also has a limit on the path length, usually 256 or 1024 bytes.
- "**" can only be at the end of the path or be followed by a path separator or by a number and a path separator.

You can combine "*" and "**" in any order:

```
/usr/**/sys/*
/usr/*tory/sys/**
/usr/**2/sys/*
```

2) Upward search:
Here you can give a directory and then search the directory tree upward for a file. You could give stop-directories to limit the upward search. The stop-directories are appended to the path (for the 'path' option) or to the filename (for the 'tags' option) with a ';'. If you want several stop-directories separate them with ';'. If you want no stop-directory ("search upward till the root directory") just use ';'.

```
/usr/include/sys;/usr
```

will search in:

```
/usr/include/sys
/usr/include
/usr
```

If you use a relative path the upward search is started in Vim's current directory or in the directory of the current file (if the relative path starts with './' and 'd' is not included in 'cpoptions').

If Vim's current path is /u/user_x/work/release and you do

```
:set path=include;/u/user_x
```

and then search for a file with gf the file is searched in:

```
/u/user_x/work/release/include
/u/user_x/work/include
/u/user_x/include
```

Note: If your 'path' setting includes a non-existing directory, Vim will skip the non-existing directory, and also does not search in the parent of the non-existing directory if upwards searching is used.

3) Combined up/downward search:
If Vim's current path is /u/user_x/work/release and you do

```
set path=**;/u/user_x
```

and then search for a file with gf the file is searched in:

```
/u/user_x/work/release/**
/u/user_x/work/**
/u/user_x/**
```

BE CAREFUL! This might consume a lot of time, as the search of '/u/user_x/**' includes '/u/user_x/work/**' and '/u/user_x/work/release/**'. So '/u/user_x/work/release/**' is searched three times and '/u/user_x/work/**' is searched twice.

In the above example you might want to set path to:

```
:set path=**,/u/user_x/**
```

This searches:

```
/u/user_x/work/release/**
/u/user_x/**
```

This searches the same directories, but in a different order.

Note that completion for ":find", ":sfind", and ":tabfind" commands do not currently work with 'path' items that contain a URL or use the double star with depth limiter (/usr/**2) or upward search (;) notations.

Trusted Files

trust

Nvim executes arbitrary code found on the filesystem if 'exrc' is enabled. To prevent executing malicious code, only "trusted files" are executed. You can mark a file as trusted or untrusted using the :trust command or the vim.secure.read() function.

:trust E5570

:trust [++deny] [++remove] [file]

Manage trusted files. Without ++ options, :trust marks the current buffer (or [file], but see WARNING below) as trusted, keyed on a hash of its contents. The trust list is stored on disk, Nvim will re-use it after restarting.

WARNING: :trust [file] has a TOCTOU risk, so for security you should view the file, inspect its contents, and run :trust (without [file] arg).

[++deny] marks the current buffer or [file] as untrusted: it will never be executed, 'exrc' will ignore it.

[++remove] removes the current buffer or [file] from the trust list. When the file is discovered by 'exrc' or vim.secure.read(), the user will be prompted to trust or deny it.

Log Files

log-files

Nvim keeps log files in stdpath("log"). See standard-path for where that location is on your system.

:log E5200

:log [logname]

Open a log file that exists in stdpath("log"). [logname] is the name of the file with the extension ".log" removed.

Without a provided [logname], opens the log directory.

Examples:

```
:log nvim
```

Open the general log file for Nvim.

```
:log lsp
```

Open the lsp-log file.

```
:log nvim-pack
```

Open the vim.pack log file.

General Questions faq-general

WHERE SHOULD I PUT MY CONFIG (VIMRC)?

See config; you can copy (or symlink) your existing vimrc. nvim-from-vim

HOW STABLE IS THE DEVELOPMENT (PRE-RELEASE) VERSION?

The unstable (pre-release) https://github.com/neovim/neovim/releases/tag/nightly version of Nvim ("HEAD", i.e. the `master` branch) is used to aggressively stage new features and changes. It's usually stable, but will occasionally break your workflow. We depend on HEAD users to report "blind spots" that were not caught by automated tests.

Use the stable (release) https://github.com/neovim/neovim/releases/latest version for a more predictable experience.

CAN I USE LUA-BASED VIM PLUGINS (E.G. NEOCOMPLETE)?

No. Starting with Nvim 0.2 PR #4411 https://github.com/neovim/neovim/pull/4411 Lua is built-in, but the legacy Vim `if_lua` interface is not supported.

HOW CAN I USE "TRUE COLOR" IN THE TERMINAL?

Truecolor (24bit colors) are enabled by default if a supporting terminal is detected. If your terminal is not detected but you are sure it supports truecolor, add this to your init.vim:

```
set termguicolors
```

NVIM SHOWS WEIRD SYMBOLS (`^[[2 q`) WHEN CHANGING MODES

This is a bug in your terminal emulator. It happens because Nvim sends cursor-shape termcodes by default, if the terminal appears to be xterm-compatible (`TERM=xterm-256color`).

To workaround the issue, you can:

- Use a different terminal emulator
- Disable 'guicursor' in your Nvim config:

```
:set guicursor=
" Workaround some broken plugins which set guicursor indiscriminately.
:autocmd OptionSet guicursor noautocmd set guicursor=
```

See also $TERM for recommended values of `$TERM`.

HOW TO CHANGE CURSOR SHAPE IN THE TERMINAL?

- For Nvim 0.1.7 or older: see the note about `NVIM_TUI_ENABLE_CURSOR_SHAPE` in `man nvim`.
- For Nvim 0.2 or newer: cursor styling is controlled by the 'guicursor' option.
  - To _disable_ cursor-styling, set 'guicursor' to empty:

    ```
    :set guicursor=
    " Workaround some broken plugins which set guicursor indiscriminately.
    :autocmd OptionSet guicursor noautocmd set guicursor=
    ```

  - If you want a non-blinking cursor, use `blinkon0`. See 'guicursor'.
  - 'guicursor' is enabled by default, unless Nvim thinks your terminal doesn't support it. If you're sure that your terminal supports cursor-shaping, set 'guicursor' in your init.vim, as described in 'guicursor'.
- The Vim terminal options t_SI and `t_EI` are ignored, like all other t_xx options.
- Old versions of libvte (gnome-terminal, roxterm, terminator, ...) do not support cursor style control codes. #2537 https://github.com/neovim/neovim/issues/2537

HOW TO CHANGE CURSOR COLOR IN THE TERMINAL?

Cursor styling (shape, color, behavior) is controlled by 'guicursor', even in the terminal. Cursor color (as opposed to shape) only works if 'termguicolors' is set.

'guicursor' gives an example, but here's a more complicated example which sets different colors in insert-mode and normal-mode:

```
:set termguicolors
:hi Cursor guifg=green guibg=green
:hi Cursor2 guifg=red guibg=red
:set guicursor=n-v-c:block-Cursor/lCursor,i-ci-ve:ver25-Cursor2/lCursor2,r-cr:hor20,o:hor50
```

CURSOR STYLE ISN'T RESTORED AFTER EXITING OR SUSPENDING AND RESUMING NVIM

Nvim uses the "Se" terminfo capability to reset the cursor style on exit. If your terminal's terminfo does not define this capability, use autocommands to manage the cursor style:

```
au VimEnter,VimResume * set guicursor=n-v-c:block,i-ci-ve:ver25,r-cr:hor20,o:hor50
  \,a:blinkwait700-blinkoff400-blinkon250-Cursor/lCursor
  \,sm:block-blinkwait175-blinkoff150-blinkon175

au VimLeave,VimSuspend * set guicursor=a:block-blinkon0
```

Alternatively, consider setting the desired cursor shape in your shell (e.g. with PROMPT_COMMAND in bash).

CURSOR SHAPE DOESN'T CHANGE IN TMUX

tmux decides that, not Nvim. See tui-cursor-shape for a fix.

See #3165 https://github.com/neovim/neovim/pull/3165 for discussion.

CURSOR FLICKER IN TMUX?

If cursor _ appears and disappears very quickly when opening nvim without a document under tmux, and you set ctermbg in `EndOfBuffer` and `Normal`, try setting these to `NONE`:

```
hi EndOfBuffer ctermbg=NONE ctermfg=200 cterm=NONE
hi Normal ctermbg=NONE ctermfg=200 cterm=NONE
```

WHAT HAPPENED TO --remote AND FRIENDS?

--remote is partly supported. clientserver

If you require flags from Vim that are missing in Nvim, you can use https://github.com/mhinz/neovim-remote instead.

Runtime issues faq-runtime

COPYING TO X11 PRIMARY SELECTION WITH THE MOUSE DOESN'T WORK

`clipboard=autoselect` is not implemented yet https://github.com/neovim/neovim/issues/2325. You may find this workaround to be useful:

```
vnoremap <LeftRelease> "*ygv
vnoremap <2-LeftRelease> "*ygv
```

MY CTRL-H MAPPING DOESN'T WORK

This was fixed in Nvim 0.2. If you are running Nvim 0.1.7 or older, adjust your terminal's "kbs" (key_backspace) terminfo entry:

```
infocmp $TERM | sed 's/kbs=^[hH]/kbs=\\177/' > $TERM.ti
tic $TERM.ti
```

(Feel free to delete the temporary `*.ti` file created after running the above commands).

<HOME> OR SOME OTHER "SPECIAL" KEY DOESN'T WORK

Make sure $TERM is set correctly.

- For screen or tmux, `$TERM` should be `screen-256color` (not `xterm-256color`!)
- In other cases if "256" does not appear in the string it's probably wrong. Try `TERM=xterm-256color`.

:! AND SYSTEM() DO WEIRD THINGS WITH INTERACTIVE PROCESSES

Interactive commands are supported by :terminal in Nvim. But :! and system() do not support interactive commands, primarily because Nvim UIs use stdio for msgpack communication, but also for performance, reliability, and consistency across platforms (see https://vimhelp.org/gui_x11.txt.html#gui-pty).

See also #1496 https://github.com/neovim/neovim/issues/1496 and #8217 https://github.com/neovim/neovim/issues/8217#issuecomment-402152307.

PYTHON SUPPORT ISN'T WORKING

Run :checkhealth in Nvim for automatic diagnosis.

Other hints:

- Read provider-python to learn how to install `pynvim`.
- Be sure you have the latest version of the `pynvim` Python module:

  ```
  uv tool install --upgrade pynvim
  ```

  See provider-python for other installation options.
- If you're manually creating a Python virtual environment for the `pynvim` module https://pypi.org/project/pynvim/, you must set `g:python3_host_prog` to the virtualenv's interpreter path.
- Try with `nvim -u NORC` to make sure your config (init.vim) isn't causing a problem. If you get `E117: Unknown function`, that means there's a runtime issue: faq-runtime.
- The python `neovim` module was renamed to `pynvim` (long ago).

:CHECKHEALTH REPORTS E5009: INVALID $VIMRUNTIME

This means $VIMRUNTIME or 'runtimepath' is broken.

- $VIMRUNTIME must point to Nvim's runtime files, not Vim's.
- The $VIMRUNTIME directory contents should be readable by the current user.
- Verify that `:echo &runtimepath` contains the $VIMRUNTIME path.

NEOVIM CAN'T FIND ITS RUNTIME

This is the case if `:help nvim` shows `E149: No help for nvim`.

Make sure that $VIM and $VIMRUNTIME point to Nvim's (as opposed to Vim's) runtime by checking `:echo $VIM` and `:echo $VIMRUNTIME`. This should give something like `/usr/share/nvim` resp. `/usr/share/nvim/runtime`.

Also make sure that you don't accidentally overwrite your runtimepath (`:set runtimepath?`), which includes the above $VIMRUNTIME by default (see 'runtimepath').

NEOVIM IS SLOW

Use a fast terminal emulator:

- kitty https://github.com/kovidgoyal/kitty
- alacritty https://github.com/jwilm/alacritty

Use an optimized build:

`:version` should report one of these "build types":

```
Build type: RelWithDebInfo
Build type: MinSizeRel
Build type: Release
```

If it reports `Build type: Debug` and you're building Nvim from source, see https://github.com/neovim/neovim/blob/master/BUILD.md.

COLORS AREN'T DISPLAYED CORRECTLY

Ensure that $TERM is set correctly.

From a shell, run `TERM=xterm-256color nvim`. If colors are displayed correctly, then export that value of `TERM` in your user profile (usually `~/.profile`):

```
export TERM=xterm-256color
```

If you're using `tmux`, instead add this to your `tmux.conf`:

```
set -g default-terminal "tmux-256color"
```

For GNU `screen`, configure your `.screenrc` <https://wiki.archlinux.org/index.php/GNU_Screen#Use_256_colors>:

```
term screen-256color
```

NOTE: Nvim ignores `t_Co` and other t_xx terminal codes.

NEOVIM CAN'T READ UTF-8 CHARACTERS

Run the following from the command line:

```
locale | grep -E '(LANG|LC_CTYPE|LC_ALL)=(.*\.)?(UTF|utf)-?8'
```

If there's no results, you might not be using a UTF-8 locale. See these issues:

- https://github.com/neovim/neovim/issues/1601
- https://github.com/neovim/neovim/issues/1858
- https://github.com/neovim/neovim/issues/2386

ESC IN TMUX OR GNU SCREEN IS DELAYED

This is a common problem https://www.google.com/?q=tmux%20vim%20escape%20delay in `tmux` / `screen` (see also https://github.com/tmux/tmux/issues/131#issuecomment-145853211). The corresponding timeout needs to be tweaked to a low value (10-20ms).

`.tmux.conf`:

```
set -g escape-time 10
# Or for tmux >= 2.6
set -sg escape-time 10
```

`.screenrc`:

```
maptimeout 10
```

"WHY DOESN'T THIS HAPPEN IN VIM?"

It does happen (try `vim -N -u NONE`), but if you hit a key quickly after ESC then Vim interprets the ESC as ESC instead of ALT (META). You won't notice the delay unless you closely observe the cursor. The tradeoff is that Vim won't understand ALT (META) key-chords, so for example `nnoremap <M-a>` won't work. ALT (META) key-chords always work in Nvim. See also `:help xterm-cursor-keys` in Vim.

Nvim 0.3 mimics the Vim behavior while still fully supporting ALT mappings. See i_ALT.

ESC IN GNU SCREEN IS LOST WHEN MOUSE MODE IS ENABLED

This happens because of a bug in screen https://savannah.gnu.org/bugs/?60196 : in mouse mode, screen assumes that ESC is part of a mouse sequence and will wait an unlimited time for the rest of the sequence, regardless of `maptimeout`. Until it's fixed in screen, there's no known workaround for this other than double-pressing escape, which causes a single escape to be passed through to Nvim.

CALLING INPUTLIST(), ECHOMSG, ... IN FILETYPE PLUGINS AND AUTOCMD DOES NOT WORK

- https://github.com/neovim/neovim/issues/10008
- https://github.com/neovim/neovim/issues/10116
- https://github.com/neovim/neovim/issues/12288
- https://github.com/vim/vim/issues/4379

This is because Nvim sets `shortmess+=F` by default. Vim behaves the same way with `set shortmess+=F`. There are plans to improve this, but meanwhile as a workaround, use `set shortmess-=F` or use `unsilent` as follows.

```
unsilent let var = inputlist(['1. item1', '2. item2'])
autocmd BufNewFile * unsilent echomsg 'The autocmd has been fired.'
```

G:CLIPBOARD SETTINGS ARE NOT USED.

If the clipboard provider is already loaded, you will need to reload it after configuration. Use the following configuration.

```
let g:clipboard = { 'name' : ... }
if exists('g:loaded_clipboard_provider')
  unlet g:loaded_clipboard_provider
  runtime autoload/provider/clipboard.vim
endif
```

Or, if you want automatic reloading when assigning to g:clipboard, set init.vim as follows.

```
function! s:clipboard_changed(...) abort
  if exists('g:loaded_clipboard_provider')
    unlet g:loaded_clipboard_provider
  endif
  runtime autoload/provider/clipboard.vim
endfunction

if !exists('s:loaded')
  call dictwatcheradd(g:, 'clipboard', function('s:clipboard_changed'))
endif
let s:loaded = v:true
```

Build issues faq-build

GENERAL BUILD ISSUES

Run `make distclean && make` to rule out a stale build environment causing the failure.

SETTINGS IN LOCAL.MK DON'T TAKE EFFECT

CMake caches build settings, so you might need to run `rm -r build && make` after modifying `local.mk`.

CMAKE ERRORS

`configure_file Problem configuring file`

This is probably a permissions issue, which can happen if you run `make` as the root user, then later run an unprivileged `make`. To fix this, run `rm -rf build` and try again.

GENERATING HELPTAGS FAILED

If re-installation fails with "Generating helptags failed", try removing the previously installed runtime directory (if `CMAKE_INSTALL_PREFIX` is not set during building, the default is `/usr/local/share/nvim`):

```
rm -r /usr/local/share/nvim
```

Design faq-design

WHY NOT USE JSON FOR RPC?

- JSON cannot easily/efficiently handle binary data
- JSON specification is ambiguous: https://seriot.ch/parsing_json.php

WHY EMBED LUA INSTEAD OF X?

- Lua is a very small language, ideal for embedding. The biggest advantage of Python/Ruby/etc is their huge collection of libraries, but that isn't relevant for Nvim, where Nvim is the "batteries included" library: introducing another stdlib would be redundant.
- Lua 5.1 is a complete language: the syntax is frozen. This is great for backwards compatibility.

- Nvim also uses Lua internally as an alternative to C. Extra performance is useful there, as opposed to a slow language like Python or Vim9script.
- LuaJIT is one of the fastest runtimes on the planet, 10x faster than Python and "Vim9script" https://vimhelp.org/vim9.txt.html , 100x faster than Vimscript.
- Python/JS cost more than Lua in terms of size and portability, and there are already numerous Python/JS-based editors. So Python/JS would make Nvim bigger and less portable, in exchange for a non-differentiating feature.

See also:

- Why Lua https://web.archive.org/web/20150219224654/https://blog.datamules.com/blog/2012/01/30/why-lua/
- The Design of Lua https://cacm.acm.org/research/a-look-at-the-design-of-lua/
- Scripting architecture considerations http://oldblog.antirez.com/post/redis-and-scripting.html
- LuaJIT performance https://julialang.org/benchmarks/
- Discussion of JavaScript vs Lua https://github.com/vim/vim/pull/5198#issuecomment-554693754
- Discussion Python embedding https://lobste.rs/s/pnuak4/mercurial_s_journey_reflections_on#c_zshdwy

WHY LUA 5.1 INSTEAD OF LUA 5.3+?

Lua 5.1 is a different language than 5.3. The Lua org makes breaking changes with every new version, so even if we switched (not upgraded, but switched) to 5.3 we gain nothing when they create the next new language in 5.4, 5.5, etc. And we would lose LuaJIT, which is far more valuable than Lua 5.3+.

Lua 5.1 is a complete language. To "upgrade" it, add libraries, not syntax. Nvim itself already is a pretty good "stdlib" for Lua, and we will continue to grow and enhance it. Changing the rules of Lua gains nothing in this context.

WILL NEOVIM TRANSLATE VIMSCRIPT TO LUA, INSTEAD OF EXECUTING VIMSCRIPT DIRECTLY?

We have no plans for transpiling Vimscript. It was explored in https://github.com/tjdevries/vim9jit

ARE PLUGIN AUTHORS ENCOURAGED TO PORT THEIR PLUGINS FROM VIMSCRIPT TO LUA? DO YOU PLAN ON SUPPORTING VIMSCRIPT INDEFINITELY?

We don't anticipate any reason to deprecate Vimscript, which is a valuable DSL https://en.wikipedia.org/wiki/Domain-specific_language for text-editing tasks. Maintaining Vimscript compatibility is less costly than a mass migration of existing Vim plugins.

Porting from Vimscript to Lua just for the heck of it gains nothing. Nvim is emphatically a fork of Vim in order to leverage the work already spent on thousands of Vim plugins, while enabling new types of plugins and integrations.

That being said, reimplementing legacy plugins in Lua in order to make use of Nvim API and to integrate with Nvim-specific features such as treesitter can be worthwhile.

Filetypes file-type

Also see autocmd.txt.

1. Filetypes filetypes file-types

Vim can detect the type of file that is edited. This is done by checking the file name and sometimes by inspecting the contents of the file for specific text.

:filetype :filet

To enable file type detection, use this command in your vimrc:

```
:filetype on
```

Each time a new or existing file is edited, Vim will try to recognize the type of the file and set the 'filetype' option. This will trigger the FileType event, which can be used to set the syntax highlighting, set options, etc.

Detail: The ":filetype on" command will load these files:
$VIMRUNTIME/filetype.lua
$VIMRUNTIME/filetype.vim
filetype.lua creates an autocommand that fires for all BufNewFile and BufRead events. It tries to detect the filetype based off of the file's extension or name.

filetype.vim is a Vim script that defines autocommands for the BufNewFile and BufRead events. In contrast to filetype.lua, this file creates separate BufNewFile and BufRead events for each filetype pattern.

If the file type is not found by the name, the file $VIMRUNTIME/scripts.vim is used to detect it from the contents of the file.
When the GUI is running or will start soon, the menu.vim script is also sourced.

To add your own file types, see new-filetype below. To search for help on a filetype prepend "ft-" and optionally append "-syntax", "-indent" or "-plugin". For example:

```
:help ft-vim-indent
:help ft-vim-syntax
:help ft-man-plugin
```

If the file type is not detected automatically, or it finds the wrong type, you can either set the 'filetype' option manually, or add a modeline to your file. Example, for an IDL file use the command:

```
:set filetype=idl
```

or add this modeline to the file:

```
/* vim: set filetype=idl : */
```

:filetype-plugin-on

You can enable loading the plugin files for specific file types with:

```
:filetype plugin on
```

If filetype detection was not switched on yet, it will be as well.
This actually loads the file "ftplugin.vim" in 'runtimepath'.
The result is that when a file is edited its plugin file is loaded (if there is one for the detected filetype). filetype-plugin

:filetype-plugin-off

You can disable it again with:

```
:filetype plugin off
```

The filetype detection is not switched off then. But if you do switch off filetype detection, the plugins will not be loaded either.
This actually loads the file "ftplugof.vim" in 'runtimepath'.

:filetype-indent-on

You can enable loading the indent file for specific file types with:

```
:filetype indent on
```

If filetype detection was not switched on yet, it will be as well.
This actually loads the file "indent.vim" in 'runtimepath'.
The result is that when a file is edited its indent file is loaded (if there is one for the detected filetype). indent-expression

:filetype-indent-off

You can disable it again with:

```
:filetype indent off
```

The filetype detection is not switched off then. But if you do switch off
filetype detection, the indent files will not be loaded either.
This actually loads the file "indoff.vim" in 'runtimepath'.
This disables auto-indenting for files you will open. It will keep working in
already opened files. Reset 'autoindent', 'cindent', 'smartindent' and/or
'indentexpr' to disable indenting in an opened file.

:filetype-off

To disable file type detection, use this command:

```
:filetype off
```

This will keep the flags for "plugin" and "indent", but since no file types
are being detected, they won't work until the next ":filetype on".

Overview: :filetype-overview

command	detection	plugin	indent
:filetype on	on	unchanged	unchanged
:filetype off	off	unchanged	unchanged
:filetype plugin on	on	on	unchanged
:filetype plugin off	unchanged	off	unchanged
:filetype indent on	on	unchanged	on
:filetype indent off	unchanged	unchanged	off
:filetype plugin indent on	on	on	on
:filetype plugin indent off	unchanged	off	off

To see the current status, type:

```
:filetype
```

The output looks something like this:

```
filetype detection:ON  plugin:ON  indent:OFF
```

The file types are also used for syntax highlighting. If the ":syntax on"
command is used, the file type detection is installed too. There is no need
to do ":filetype on" after ":syntax on".

To disable one of the file types, add a line in your filetype file, see
remove-filetype.

filetype-detect

To detect the file type again:

```
:filetype detect
```

Use this if you started with an empty file and typed text that makes it
possible to detect the file type. For example, when you entered this in a
shell script: "#!/bin/csh".
   When filetype detection was off, it will be enabled first, like the "on"
argument was used.

filetype-overrule

When the same extension is used for multiple filetypes, Vim tries to guess
what kind of file it is. This doesn't always work. A number of global
variables can be used to overrule the filetype used for certain extensions:

file name	variable		
*.al	g:filetype_al		
*.app	g:filetype_app		
*.as	g:filetype_as		
*.asa	g:filetype_asa	ft-aspperl-syntax ft-aspvbs-syntax	
*.asm	g:asmsyntax	ft-asm-syntax	
*.asp	g:filetype_asp	ft-aspperl-syntax ft-aspvbs-syntax	
*.bas	g:filetype_bas	ft-basic-syntax	
*.cfg	g:filetype_cfg		
*.cls	g:filetype_cls		
*.csh	g:filetype_csh	ft-csh-syntax	
*.dat	g:filetype_dat		
*.def	g:filetype_def		
*.dsp	g:filetype_dsp		
*.f	g:filetype_f	ft-forth-syntax	
*.frm	g:filetype_frm	ft-form-syntax	
*.fs	g:filetype_fs	ft-forth-syntax	
*.h	g:c_syntax_for_h	ft-c-syntax	(deprecated)
*.h	g:ch_syntax_for_h	ft-ch-syntax	(deprecated)
*.h	g:filetype_h	ft-c-syntax ft-ch-syntax ft-cpp-syntax	
*.i	g:filetype_i	ft-progress-syntax	
*.inc	g:filetype_inc		
*.int	g:filetype_int		
*.lsl	g:filetype_lsl		
*.m	g:filetype_m	ft-mathematica-syntax	
*.mm	g:filetype_mm		
*.mac	g:filetype_mac *[mM]makefile,*.mk,*.mak,[mM]akefile*		
g:make_flavor	ft-make-syntax *.markdown,*.mdown,*.mkd,*.mkdn,*.mdwn,*.md		
g:filetype_md	ft-pandoc-syntax		
*.mod	g:filetype_mod		
*.p	g:filetype_p	ft-pascal-syntax	
*.pl	g:filetype_pl		
*.pp	g:filetype_pp	ft-pascal-syntax	
*.prg	g:filetype_prg		
*.r	g:filetype_r		
*.sig	g:filetype_sig		
*.sql	g:filetype_sql	ft-sql-syntax	
*.src	g:filetype_src		
*.sys	g:filetype_sys		
*.sh	g:bash_is_sh	ft-sh-syntax	
*.tex	g:tex_flavor	ft-tex-plugin	
*.tf	g:filetype_tf		
*.typ	g:filetype_typ		
*.v	g:filetype_v		
*.w	g:filetype_w	ft-cweb-syntax	

For a few filetypes the global variable is used only when the filetype could
not be detected:

*.r	g:filetype_r	ft-rexx-syntax	

filetype-ignore

To avoid that certain files are being inspected, the g:ft_ignore_pat variable
is used. The default value is set like this:

```
:let g:ft_ignore_pat = '\.\(Z\|gz\|bz2\|zip\|tgz\)$'
```

This means that the contents of compressed files are not inspected.

new-filetype

If a file type that you want to use is not detected yet, there are a few ways
to add it. The recommended way is to use vim.filetype.add() to add it to
Nvim's builtin filetype detection mechanism. If you want to handle the
detection manually, proceed as follows:

A. If you want to overrule all default file type checks.
   This works by writing one file for each filetype. The disadvantage is that
   there can be many files. The advantage is that you can simply drop this
   file in the right directory to make it work.

ftdetect

 1. Create your user runtime directory. You would normally use the first

       item of the 'runtimepath' option. Then create the directory "ftdetect"
       inside it. Example for Unix:

```
:!mkdir -p ~/.config/nvim/ftdetect
```

 2. Create a file that contains an autocommand to detect the file type.

       Example:

```
au BufRead,BufNewFile *.mine            set filetype=mine
```

      Note that there is no "augroup" command, this has already been done
       when sourcing your file. You could also use the pattern "*" and then
       check the contents of the file to recognize it.
       Write this file as "mine.vim" in the "ftdetect" directory in your user
       runtime directory. For example, for Unix:

```
:w ~/.config/nvim/ftdetect/mine.vim
```

 3. To use the new filetype detection you must restart Vim.

   The files in the "ftdetect" directory are used after all the default
   checks, thus they can overrule a previously detected file type. But you
   can also use :setfiletype to keep a previously detected filetype.

B. If you want to detect your file after the default file type checks.

   This works like A above, but instead of setting 'filetype' unconditionally
   use ":setfiletype". This will only set 'filetype' if no file type was
   detected yet. Example:

```
au BufRead,BufNewFile *.txt             setfiletype text
```

   You can also use the already detected file type in your command. For
   example, to use the file type "mypascal" when "pascal" has been detected:

```
au BufRead,BufNewFile *         if &ft == 'pascal' | set ft=mypascal
                                                     | endif
```

C. If your file type can be detected by the file name or extension.
 1. Create your user runtime directory. You would normally use the first

       item of the 'runtimepath' option. Example for Unix:

```
:!mkdir -p ~/.config/nvim
```

 2. Create a file that contains autocommands to detect the file type.

       Example:

```
" my filetype file
if exists("did_load_filetypes")
  finish
endif
augroup filetypedetect
  au! BufRead,BufNewFile *.mine         setfiletype mine
  au! BufRead,BufNewFile *.xyz          setfiletype drawing
augroup END
```

       Write this file as "filetype.vim" in your user runtime directory. For
       example, for Unix:

```
:w ~/.config/nvim/filetype.vim
```

 3. To use the new filetype detection you must restart Vim.

   Your filetype.vim will be sourced before the default FileType autocommands
   have been installed. Your autocommands will match first, and the
   ":setfiletype" command will make sure that no other autocommands will set

'filetype' after this.

new-filetype-scripts

D. If your filetype can only be detected by inspecting the contents of the file.

1. Create your user runtime directory. You would normally use the first item of the 'runtimepath' option. Example for Unix:

```
:!mkdir -p ~/.config/nvim
```

2. Create a Vim script file for doing this. Example:

```
if did_filetype()       " filetype already set..
  finish                " ..don't do these checks
endif
if getline(1) =~ '^#!.*\<mine\>'
  setfiletype mine
elseif getline(1) =~? '\<drawing\>'
  setfiletype drawing
endif
```

See $VIMRUNTIME/scripts.vim for more examples.
Write this file as "scripts.vim" in your user runtime directory. For example, for Unix:

```
:w ~/.config/nvim/scripts.vim
```

3. The detection will work right away, no need to restart Vim.

Your scripts.vim is loaded before the default checks for file types, which means that your rules override the default rules in $VIMRUNTIME/scripts.vim.

remove-filetype

If a file type is detected that is wrong for you, you can set 'filetype' to a non-existing name such as ignored to avoid that it will be set later anyway.

g:did_load_filetypes

The builtin filetype detection provided by Nvim can be disabled by setting the did_load_filetypes global variable. If this variable exists, the default $VIMRUNTIME/filetype.lua will not run.

plugin-details

The "plugin" directory can be in any of the directories in the 'runtimepath' option. All of these directories will be searched for plugins and they are all loaded. For example, if this command:

```
set runtimepath
```

produces this output:

```
runtimepath=/etc/vim,~/.config/nvim,/usr/local/share/vim/vim82
```

then Vim will load all plugins in these directories and below:

```
/etc/vim/plugin/
~/.config/nvim/plugin/
/usr/local/share/vim/vim82/plugin/
```

Note that the last one is the value of $VIMRUNTIME which has been expanded.

Note that when using a plugin manager or packages many directories will be added to 'runtimepath'. These plugins each require their own directory, don't put them directly in ~/.config/nvim/plugin.

What if it looks like your plugin is not being loaded? You can find out what happens when Vim starts up by using the -V argument:

```
vim -V2
```

You will see a lot of messages, in between them is a remark about loading the plugins. It starts with:

```
Searching for "plugin/**/*.vim" in
```

There you can see where Vim looks for your plugin scripts.

2. Filetype plugin

filetype-plugins

When loading filetype plugins has been enabled :filetype-plugin-on, options
will be set and mappings defined. These are all local to the buffer, they
will not be used for other files.

Defining mappings for a filetype may get in the way of the mappings you
define yourself. There are a few ways to avoid this:

1. Set the "maplocalleader" variable to the key sequence you want the mappings to start with. Example:

```
:let maplocalleader = ","
```

   All mappings will then start with a comma instead of the default, which is a backslash. Also see <LocalLeader>.
2. Define your own mapping. Example:

```
:map ,p <Plug>MailQuote
```

   You need to check the description of the plugin file below for the functionality it offers and the string to map to.
   You need to define your own mapping before the plugin is loaded (before editing a file of that type). The plugin will then skip installing the default mapping.

no_mail_maps g:no_mail_maps

3. Disable defining mappings for a specific filetype by setting a variable, which contains the name of the filetype. For the "mail" filetype this would be:

```
:let no_mail_maps = 1
```

no_plugin_maps g:no_plugin_maps

4. Disable defining mappings for all filetypes by setting a variable:

```
:let no_plugin_maps = 1
```

ftplugin-overrule

If a global filetype plugin does not do exactly what you want, there are three
ways to change this:

1. Add a few settings.

   You must create a new filetype plugin in a directory early in 'runtimepath'. For Unix, for example you could use this file:

```
vim ~/.config/nvim/ftplugin/fortran.vim
```

   You can set those settings and mappings that you would like to add. Note that the global plugin will be loaded after this, it may overrule the settings that you do here. If this is the case, you need to use one of the following two methods.
2. Make a copy of the plugin and change it.

   You must put the copy in a directory early in 'runtimepath'. For Unix, for example, you could do this:

```
cp $VIMRUNTIME/ftplugin/fortran.vim ~/.config/nvim/ftplugin/fortran.vim
```

   Then you can edit the copied file to your liking. Since the b:did_ftplugin variable will be set, the global plugin will not be loaded.
   A disadvantage of this method is that when the distributed plugin gets improved, you will have to copy and modify it again.
3. Overrule the settings after loading the global plugin.

   You must create a new filetype plugin in a directory from the end of 'runtimepath'. For Unix, for example, you could use this file:

```
vim ~/.config/nvim/after/ftplugin/fortran.vim
```

   In this file you can change just those settings that you want to change.

3. Docs for the default filetype plugins.

ftplugin-docs

plugin_exec g:plugin_exec

Enable executing of external commands. This was done historically for e.g.
the perl filetype plugin (and a few others) to set the search path.
Disabled by default for security reasons:

```
:let g:plugin_exec = 1
```

It is also possible to enable this only for certain filetypes:

```
:let g:<filetype>_exec = 1
```

So to enable this only for ruby, set the following variable:

```
:let g:ruby_exec = 1
```

If both, the global plugin_exec and the <filetype>_exec specific variable
are set, the filetype specific variable should have precedent.

ADA

Documentation for this plugin is here: ft_ada.txt.

ASCIIDOC ft-asciidoc-plugin

To enable folding use this:

```
let g:asciidoc_folding = 1
```

To disable nesting of folded headers use this:

```
let g:asciidoc_foldnested = 0
```

To disable folding everything under the title use this:

```
let asciidoc_fold_under_title = 0
```

ARDUINO ft-arduino-plugin

By default the following options are set, in accordance with the default
settings of Arduino IDE:

```
setlocal expandtab tabstop=2 softtabstop=2 shiftwidth=2
```

To disable this behavior, set the following variable in your vimrc:

```
let g:arduino_recommended_style = 0
```

AWK ft-awk-plugin

Support for features specific to GNU Awk, like @include, can be enabled by
setting:

```
:let g:awk_is_gawk = 1
```

BEANCOUNT ft-beancount-plugin

Beancount omni-completion compl-omni is provided by beancountcomplete.vim,
if enabled with g:beancount_completion_enable.

To enable completion of account names, set:

```
let g:beancount_completion_enable = 1
```

Note: Enabling this may cause beancount to load additional plugin files. Only
enable for code you trust.

Variables:

g:beancount_account_completion
Specify the completion pattern format for the account
name. Can be either "default" or "chunks".
Default value: "default"

g:beancount_completion_enable
Enable omni-completion compl-omni for accounts.
Default value: 0

g:beancount_detailed_first
If non-zero, accounts higher down the hierarchy will
be listed first as completions.
Default value: 0

g:beancount_separator_col
The column that the decimal separator is aligned to.
Default value: 50

b:beancount_root
Set the root Beancount file. This is used to gather

values for the completion.
If not set (default), the current file will be used.

Commands:

:AlignCommodity — Adds spaces between an account and commodity so that the decimal points of the commodities all occur in the column given by g:beancount_separator_col. If an amount has no decimal point, the imaginary decimal point to the right of the least significant digit will align.

The command acts on a range, with the default being the current line. If the cursor happens to be inside that range and to the right of the account name, the cursor will be pushed to the right the appropriate amount, so that it remains on the same character.

The script assumes the use of spaces for alignment. It does not understand tabs.

:GetContext — Uses bean-doctor context to display the context of the current line.

CHANGELOG ft-changelog-plugin

Allows for easy entrance of Changelog entries in Changelog files. There are some commands, mappings, and variables worth exploring:

Options:

'comments' — is made empty to not mess up formatting.
'textwidth' — is set to 78, which is standard.
'formatoptions' — the 't' flag is added to wrap when inserting text.

Commands:

NewChangelogEntry — Adds a new Changelog entry in an intelligent fashion (see below).

Local mappings:

<Leader>o — Starts a new Changelog entry in an equally intelligent fashion (see below).

Global mappings:

NOTE: The global mappings are accessed by sourcing the ftplugin/changelog.vim file first, e.g. with

```
runtime ftplugin/changelog.vim
```

in your init.vim.

<Leader>o — Switches to the ChangeLog buffer opened for the current directory, or opens it in a new buffer if it exists in the current directory. Then it does the same as the local <Leader>o described above.

Variables:

g:changelog_timeformat — **Deprecated**; use g:changelog_dateformat instead.
g:changelog_dateformat — The date (and time) format used in ChangeLog entries. The format accepted is the same as for the strftime() function. The default is "%Y-%m-%d" which is the standard format for many ChangeLog layouts.
g:changelog_username — The name and email address of the user. The default is deduced from environment variables and system files. It searches /etc/passwd for the comment part of the current user, which informally contains the real name of the user up to the first separating comma. then it checks the $NAME environment variable and finally runs whoami and hostname to build an email address. The final form is

```
Full Name  <user@host>
```

g:changelog_new_date_format — The format to use when creating a new date-entry. The following table describes special tokens in the string:

%%	insert a single '%' character
%d	insert the date from above
%u	insert the user from above
%p	insert result of b:changelog_entry_prefix
%c	where to position cursor when done

The default is "%d %u\n\n\t* %p%c\n\n", which produces something like (| is where cursor will be, unless at the start of the line where it denotes the beginning of the line)

```
|2003-01-14  Full Name  <user@host>
|
|        * prefix|
```

g:changelog_new_entry_format — The format used when creating a new entry. The following table describes special tokens in the string:

%p	insert result of

```
                                b:changelog_entry_prefix
                        %c      where to position cursor when done
                    The default is "\t*%c", which produces something
                    similar to

|        * prefix|

g:changelog_date_entry_search
                    The search pattern to use when searching for a
                    date-entry.
                    The same tokens that can be used for
                    g:changelog_new_date_format can be used here as well.
                    The default is '^\s*%d\_s*%u' which finds lines
                    matching the form

|2003-01-14  Full Name  <user@host>

                    and some similar formats.

g:changelog_date_end_entry_search
                    The search pattern to use when searching for the end
                    of a date-entry.
                    The same tokens that can be used for
                    g:changelog_new_date_format can be used here as well.
                    The default is '^\s*$' which finds lines that contain
                    only whitespace or are completely empty.

b:changelog_name                                                    b:changelog_name

                    Name of the ChangeLog file to look for.
                    The default is 'ChangeLog'.

b:changelog_path
                    Path of the ChangeLog to use for the current buffer.
                    The default is empty, thus looking for a file named
                    b:changelog_name in the same directory as the
                    current buffer.  If not found, the parent directory of
                    the current buffer is searched.  This continues
                    recursively until a file is found or there are no more
                    parent directories to search.

b:changelog_entry_prefix
                    Name of a function to call to generate a prefix to a
                    new entry.  This function takes no arguments and
                    should return a string containing the prefix.
                    Returning an empty prefix is fine.
                    The default generates the shortest path between the
                    ChangeLog's pathname and the current buffers pathname.
                    In the future, it will also be possible to use other
                    variable contexts for this variable, for example, g:.

The Changelog entries are inserted where they add the least amount of text.
After figuring out the current date and user, the file is searched for an
entry beginning with the current date and user and if found adds another item
under it.  If not found, a new entry and item is prepended to the beginning of
the Changelog.
```

DIRECTORY **ft-directory**

The "directory" filetype is used for directory listing buffers, such as those opened by :edit with a directory path. See dir for mappings and behavior.

Directory entries end in "/" and use the hl-Directory highlight group.

FORTRAN **ft-fortran-plugin**

Options:

'expandtab'	is switched on to avoid tabs as required by the Fortran standards unless the user has set fortran_have_tabs in vimrc.
'textwidth'	is set to 80 for fixed source format whereas it is set to 132 for free source format. Setting the fortran_extended_line_length variable increases the width to 132 for fixed source format.
'formatoptions'	is set to break code and comment lines and to preserve long lines. You can format comments with gq.

For further discussion of fortran_have_tabs and the method used for the detection of source format see ft-fortran-syntax.

FREEBASIC **ft-freebasic-plugin**

This plugin aims to treat the four FreeBASIC dialects, "fb", "qb", "fblite" and "deprecated", as distinct languages.

The dialect will be set to the first name found in g:freebasic_forcelang, any #lang directive or $lang metacommand in the file being edited, or finally g:freebasic_lang. These global variables conceptually map to the fbc options -forcelang and -lang. If no dialect is explicitly specified "fb" will be used.

For example, to set the dialect to a default of "fblite" but still allow for any #lang directive overrides, use the following command:

```
let g:freebasic_lang = "fblite"
```

GDSCRIPT

ft-gdscript-plugin

By default the following options are set, based on Godot official docs:

```
setlocal noexpandtab softtabstop=0 shiftwidth=0
```

To disable this behavior, set the following variable in your vimrc:

```
let g:gdscript_recommended_style = 0
```

GIT COMMIT

ft-gitcommit-plugin

One command, :DiffGitCached, is provided to show a diff of the current commit
in the preview window. It is equivalent to calling "git diff --cached" plus
any arguments given to the command.

The length of the first line of the commit message used for
syntax highlighting can be configured via g:gitcommit_summary_length.
The default is 50. Example:

```
let g:gitcommit_summary_length = 70
```

GIT REBASE

ft-gitrebase-plugin

The gitrebase filetype defines the following buffer-local commands, to help
with interactive git rebase:

```
:Drop   " to discard this commit
:Edit   " to stop for editing this commit
:Fixup  " to squash (but discard the message) into the previous one
:Pick   " to pick this commit (the cursor is on)
:Reword " to pick this commit, but change the commit message
:Squash " to squash this commit into the previous one
```

In addition, the following command can be used to cycle between the different
possibilities:

```
:Cycle  " to cycle between the previous commands
```

The :Cycle command is also mapped to the CTRL-A and CTRL-X keys.
For details, see git-rebase --help.

GLEAM

ft-gleam-plugin

By default the following options are set for the recommended gleam style:

```
setlocal expandtab shiftwidth=2 softtabstop=2
```

To disable this behavior, set the following variable in your vimrc:

```
let g:gleam_recommended_style = 0
```

GO

ft-go-plugin

By default the following options are set, based on Golang official docs:

```
setlocal noexpandtab softtabstop=0 shiftwidth=0
```

To disable this behavior, set the following variable in your vimrc:

```
let g:go_recommended_style = 0
```

GPROF

ft-gprof-plugin

The gprof filetype plugin defines a mapping <C-]> to jump from a function
entry in the gprof flat profile or from a function entry in the call graph
to the details of that function in the call graph.

The mapping can be disabled with:

```
let g:no_gprof_maps = 1
```

HARE

ft-hare

Since the text for this plugin is rather long it has been put in a separate
file: ft_hare.txt.

HTML ft-html-plugin

Tag folding poses a few difficulties. Many elements, e.g. blockquote, are always delimited by start and end tags; end tags for some elements, e.g. p, can be omitted in certain contexts; void elements, e.g. hr, have no end tag. Although the rules for supporting omissible end tags are ad-hoc and involved [0], they apply to elements in scope. Assuming syntactical wellformedness, an end tag can be associated with its nearest matching start tag discoverable in scope [1] and towards the beginning of a file, whereas all unbalanced tags and inlined tags can be disregarded. Having syntax highlighting in effect, tag folding using the fold-expr method can be enabled with:

```
let g:html_expr_folding = 1
```

By default, tag folding will be redone from scratch after each occurrence of a TextChanged or an InsertLeave event. Such frequency may not be desired, especially for large files, and this recomputation can be disabled with:

```
let g:html_expr_folding_without_recomputation = 1
doautocmd FileType
```

To force another recomputation, do:

```
unlet! b:foldsmap
normal zx
```

[0] https://html.spec.whatwg.org/multipage/syntax.html#optional-tags
[1] https://en.wikipedia.org/wiki/Dangling_else

ICALENDAR ft-icalendar-plugin g:icalendar_folding

You can enable folding of iCalendar components (BEGIN: and END: pairs) using fold-expr by:

```
let g:icalendar_folding = 1
```

Nested components, such as a VALARM inside a VEVENT, produce nested folds.

IDRIS2 ft-idris2-plugin

By default the following options are set:

```
setlocal shiftwidth=2 tabstop=2 expandtab
setlocal comments=s1:{-,mb:-,ex:-},:\|\|\|,:--
setlocal commentstring=--\ %s
setlocal wildignore+=*.ibc
```

To use tabs instead of spaces for indentation, set the following variable in your vimrc:

```
let g:idris2#allow_tabchar = 1
```

JAVA ft-java-plugin

g:ftplugin_java_source_path

Whenever the variable "g:ftplugin_java_source_path" is defined and its value is a filename whose extension is either ".jar" or ".zip", e.g.:

```
let g:ftplugin_java_source_path = '/path/to/src.jar'
let g:ftplugin_java_source_path = '/path/to/src.zip'
```

and a zip plugin has already been sourced, the gf command can be used to open the selected entry.

Note that the effect of using the "gf" command WITHIN a buffer loaded with the Zip plugin depends on the version of the Zip plugin. For the Zip plugin versions that do not support Jar type archives, consider creating symbolic links with the ".zip" extension for each Jar archive of interest and assigning any such file to the variable from now on.

Otherwise, for the defined variable "g:ftplugin_java_source_path", the local value of the 'path' option will be further modified by prefixing the value of the variable, e.g.:

```
let g:ftplugin_java_source_path = $JDK_SRC_PATH
let &l:path = g:ftplugin_java_source_path .. ',' .. &l:path
```

and the "gf" command can be used on a fully-qualified type to look for a file in the "path" and to try to load it.

Remember to manually trigger the FileType event from a buffer with a Java file loaded in it each time after assigning a new value to the variable:

```
doautocmd FileType
```

Markdown documentation comments may contain common runs of vertical leading whitespace following the comment marks (////) for aesthetic reasons; however, some horizontal runs of leading whitespace are significant in Markdown because

they denote code blocks etc. For convenience, a 'formatexpr' function is
provided for the gq operator. As long as neither "g:java_ignore_javadoc"
nor "g:java_ignore_markdown" is defined, the reformatting of Markdown comments
can be enabled on demand with:

```
setlocal formatexpr=g:javaformat#RemoveCommonMarkdownWhitespace()
```

Or for Vim versions less than 7.4.265, with:

```
setlocal formatexpr=javaformat#RemoveCommonMarkdownWhitespace()
```

This function accepts a range of lines, removes a common run of vertical
leading whitespace, and rewrites the lines of the range. Depending on the
author's layout style and the comment contents, which lines to select for
reformatting can vary from the whole comment to only some portion of it.
To enable the recognition of Markdown comments each time after removing
"g:java_ignore_markdown" or "g:java_ignore_javadoc", remember to manually
re-source "javaformat.vim" for Vim versions greater than 8.2.1397:

```
runtime autoload/javaformat.vim
```

LUA **ft-lua-plugin**

g:lua_version g:lua_subversion

Lua filetype's 'includeexpr' and ft-lua-syntax highlighting use the global
variables "g:lua_version" and "g:lua_subversion" to determine the version of
Lua to use (5.3 is the default)

For example, to use Lua 5.1, set the variables like this:

```
let g:lua_version = 5
let g:lua_subversion = 1
```

LUAU **ft-luau-plugin** **g:luau_folding**

You can enable folding of Luau block constructs using fold-expr by:

```
let g:luau_folding = 1
```

MAIL **ft-mail-plugin**

Options:
'modeline'   is switched off to avoid the danger of trojan horses, and to
                avoid that a Subject line with "Vim:" in it will cause an
                error message.
'textwidth'   is set to 72. This is often recommended for e-mail.
'formatoptions' is set to break text lines and to repeat the comment leader
                in new lines, so that a leading ">" for quotes is repeated.
                You can also format quoted text with gq.

Local mappings:
<LocalLeader>q   or   \MailQuote
        Quotes the text selected in Visual mode, or from the cursor position
        to the end of the file in Normal mode. This means "> " is inserted in
        each line.

MAN **ft-man-plugin** **:Man** **man.lua**

View manpages in Nvim. Supports highlighting, completion, locales, and
navigation. Also see find-manpage.

man.lua will always attempt to reuse the closest man window (above/left) but
otherwise create a split.

The case sensitivity of completion is controlled by 'fileignorecase'.

Commands:
Man {name}   Display the manpage for {name}.
Man {sect} {name}   Display the manpage for {name} and section {sect}.
Man {name}({sect})   Same as above.
Man {sect} {name}({sect}) Used during completion to show the real section of
                          when the provided section is a prefix, e.g. 1m vs 1.
Man {path}   Open the manpage at {path}. Prepend "./" if {path}
                          is relative to the current directory.
Man   Open the manpage for the <cWORD> (man buffers)
                          or <cword> (non-man buffers) under the cursor.
Man!   Display the current buffer contents as a manpage.

:Man accepts command modifiers. For example, to use a vertical split:

```
:vertical Man printf
```

To reuse the current window:

```
:hide Man printf
```

Local mappings:
K or CTRL-]   Jump to the manpage for the <cWORD> under the
                          cursor. Takes a count for the section.
CTRL-T   Jump back to the location that the manpage was
                          opened from.

gO   Show the manpage outline. gO
q   :quit if invoked as $MANPAGER, otherwise :close.

Variables:
g:no_man_maps   Do not create mappings in manpage buffers.
g:ft_man_folding_enable Fold manpages with foldmethod=indent foldnestmax=1.
b:man_default_sects   Comma-separated, ordered list of preferred sections.
                        For example in C one usually wants section 3 or 2:

```
:let b:man_default_sections = '3,2'
```

g:man_hardwrap   Hard-wrap to $MANWIDTH or window width if $MANWIDTH is
                        empty or larger than the window width. Enabled by
                        default. Set FALSE to enable soft wrapping.

To use Nvim as a manpager:

```
export MANPAGER='nvim +Man!'
```

Note: when running man from the shell with Nvim as $MANPAGER, man will
pre-format the manpage using groff, and Nvim will display the manual page as
it was received from stdin (it can't "undo" the hard-wrap caused by
man/groff). To prevent man/groff from hard-wrapping the manpage, you can set
$MANWIDTH=999 in your environment.

To disable bold highlighting:

```
:highlight link manBold Normal
```

Troubleshooting:

If you see an error like:

```
fuse: mount failed: Permission denied
```

this may be caused by AppArmor sandboxing. To fix this, add a local override
in e.g. /etc/apparmor.d/local/usr.bin.man:

```
mount fstype=fuse.nvim options=(ro, nosuid, nodev) -> /tmp/**,
/usr/bin/fusermount Ux,
```

See also https://github.com/neovim/neovim/issues/30268 .

MARKDOWN ft-markdown-plugin

To enable folding use this:

```
let g:markdown_folding = 1
```

'expandtab' will be set by default. If you do not want that use this:

```
let g:markdown_recommended_style = 0
```

ORG ft-org-plugin

To enable folding use this:

```
let g:org_folding = 1
```

PDF ft-pdf-plugin

Two maps, <C-]> and <C-T>, are provided to simulate a tag stack for navigating
the PDF. The following are treated as tags:

- The byte offset after "startxref" to the xref table
- The byte offset after the /Prev key in the trailer to an earlier xref table
- A line of the form "0123456789 00000 n" in the xref table
- An object reference like "1 0 R" anywhere in the PDF

These maps can be disabled with

```
:let g:no_pdf_maps = 1
```

PLSQL

ft-plsql-plugin

To enable syntax folding in PL/SQL filetypes, set the following variable:

```
:let g:plsql_fold = 1
```

PYTHON

ft-python-plugin PEP8

By default the following options are set, in accordance with PEP8:

```
setlocal expandtab shiftwidth=4 softtabstop=4 tabstop=8
```

To disable this behavior, set the following variable in your vimrc:

```
let g:python_recommended_style = 0
```

Python omni-completion compl-omni is provided by python3complete.vim (or
pythoncomplete.vim) for Nvim with the python3 provider.
By default it does not inspect the import / from statements found in the
buffer. This means completion of names defined in the buffer itself (classes,
functions, variables) works, but completion of members of imported modules is
not offered.

To enable completion of imported module members, set:

```
let g:pythoncomplete_allow_import = 1
```

WARNING: enabling this causes omni-completion to execute the import statements
found in the buffer through Python's import machinery, which runs the imported
modules' top-level code. Only enable this for code you trust.

POWERSHELL

Documentation for this plugin is here: ft_ps1.txt.

QUERY

ft-query-plugin

Linting of treesitter queries for installed parsers using
vim.treesitter.query.lint() is disabled by default. To enable
linting when opening and writing buffers, add

```
vim.g.query_lint_on = { 'BufEnter', 'BufWrite' }
```

to your config. To update diagnostics when editing, use

```
vim.g.query_lint_on = { 'InsertLeave', 'TextChanged' }
```

For a dedicated language server, see https://github.com/ribru17/ts_query_ls.

QF QUICKFIX

qf.vim ft-qf-plugin

The "qf" filetype is used for the quickfix window, see quickfix-window.

The quickfix filetype plugin includes configuration for displaying the command
that produced the quickfix list in the status-line. To disable this setting,
configure as follows:

```
:let g:qf_disable_statusline = 1
```

R MARKDOWN

ft-rmd-plugin

By default ftplugin/html.vim is not sourced. If you want it sourced, add to
your vimrc:

```
let rmd_include_html = 1
```

The 'formatexpr' option is set dynamically with different values for R code
and for Markdown code. If you prefer that 'formatexpr' is not set, add to
your vimrc:

```
let rmd_dynamic_comments = 0
```

R RESTRUCTURED TEXT

ft-rrst-plugin

The 'formatexpr' option is set dynamically with different values for R code
and for ReStructured text. If you prefer that 'formatexpr' is not set, add to
your vimrc:

```
let rrst_dynamic_comments = 0
```

RESTRUCTUREDTEXT

ft-rst-plugin

The following formatting setting are optionally available:

```
setlocal expandtab shiftwidth=3 softtabstop=3 tabstop=8
```

To enable this behavior, set the following variable in your vimrc:

```
let g:rst_style = 1
```

RNOWEB ft-rnoweb-plugin

The 'formatexpr' option is set dynamically with different values for R code and for LaTeX code. If you prefer that 'formatexpr' is not set, add to your vimrc:

```
let rnw_dynamic_comments = 0
```

RAKU

Documentation for this plugin is here: ft_raku.txt.

RPM SPEC ft-spec-plugin

Since the text for this plugin is rather long it has been put in a separate file: pi_spec.txt.

SHADA ft-shada

Allows editing binary shada-files in a nice way. Opened binary files are displayed in the following format:

```
Type with timestamp YYYY-mm-ddTHH:MM:SS:
  % Key__  Description___  Value
  + fooba  foo bar baz fo  {msgpack-value}
  + abcde  abc def ghi jk  {msgpack-value}
Other type with timestamp YYYY-mm-ddTHH:MM:SS:
  @ Description__  Value
  - foo bar baz t  {msgpack-value}
  # Expected more elements in list
Some other type with timestamp YYYY-mm-ddTHH:MM:SS:
  # Unexpected type: type instead of map
  = {msgpack-value}
```

Filetype plugin defines all Cmd-events. Defined SourceCmd event makes "source file.shada" be equivalent to "|:rshada| file.shada". BufWriteCmd, FileWriteCmd and FileAppendCmd events are affected by the following settings:

g:shada#keep_old_header — Boolean, if set to false all header entries are ignored when writing. Defaults to 1.

g:shada#add_own_header — Boolean, if set to true first written entry will always be header entry with two values in a map with attached data: v:version attached to "version" key and "shada.vim" attached to "generator" key. Defaults to 1.

Format description:

1. # starts a comment. Lines starting with space characters and then # are ignored. Plugin may only add comment lines to indicate some errors in ShaDa format. Lines containing no non-whitespace characters are also ignored.
2. Each entry starts with line that has format "{type} with timestamp {timestamp}:". {timestamp} is strftime()-formatted string representing actual Unix timestamp value. First strftime() argument is equal to %Y-%m-%dT%H:%M:%S. When writing this timestamp is parsed using msgpack#strptime(), with caching (it remembers which timestamp produced particular strftime() output and uses this value if you did not change timestamp). {type} is one of
   - 1 - Header
   - 2 - Search pattern
   - 3 - Replacement string
   - 4 - History entry
   - 5 - Register
   - 6 - Variable
   - 7 - Global mark
   - 8 - Jump
   - 9 - Buffer list
   - 10 - Local mark
   - 11 - Change
   - * - Unknown (0x{type-hex})

   Each type may be represented using Unknown entry: "Jump with timestamp ..." is the same as "Unknown (0x8) with timestamp ....".

1. After header there is one of the following lines:

    2. " % Key__ Description__ Value": map header. After mapping header

            follows a table which may contain comments and lines consisting of
            " +", key, description and {msgpack-value}. Key is separated by at
            least two spaces with description, description is separated by at least
            two spaces with the value. Each key in the map must be at most as wide
            as "Key__" header: "Key" allows at most 3-byte keys, "Key__" allows at
            most 5-byte keys. If keys actually occupy less bytes then the rest is
            filled with spaces. Keys cannot be empty, end with spaces, contain two
            consequent spaces inside of them or contain multibyte characters (use
            "=" format if you need this). Descriptions have the same restrictions
            on width and contents, except that empty descriptions are allowed.
            Description column may be omitted.

        When writing description is ignored. Keys with values msgpack#equal
        to default ones are ignored. Order of keys is preserved. All keys are
        treated as strings (not binary strings).

        Note: specifically for buffer list entries it is allowed to have more
        then one map header. Each map header starts a new map entry inside
        buffer list because ShaDa buffer list entry is an array of maps. I.e.

```
Buffer list with timestamp 1970-01-01T00:00:00:
  % Key  Description  Value
  + f    file name    "/foo"
  + l    line number  1
  + c    column       10
```

        is equivalent to

```
Buffer list with timestamp 1970-01-01T00:00:00:
  = [{="f": "/foo", ="c": 10}]
```

        and

```
Buffer list with timestamp 1970-01-01T00:00:00:
  % Key  Description  Value
  + f    file name    "/foo"

  % Key  Description  Value
  + f    file name    "/bar"
```

        is equivalent to

```
Buffer list with timestamp 1970-01-01T00:00:00:
  = [{="f": "/foo"}, {="f": "/bar"}]
```

        Note 2: specifically for register entries special syntax for arrays was
        designed:

```
Register with timestamp 1970-01-01T00:00:00:
  % Key  Description  Value
  + rc   contents     @
  | - "line1"
  | - "line2"
```

        This is equivalent to

```
Register with timestamp 1970-01-01T00:00:00:
  % Key  Description  Value
  + rc   contents     ["line1", "line2"]
```

        Such syntax is automatically used if array representation appears to be
        too lengthy.
1. " @ Description__ Value": array header. Same as map, but key is

            omitted and description cannot be omitted. Array entries start with
            " -". Example:

```
History entry with timestamp 1970-01-01T00:00:00:
  @ Description_  Value
  - history type  SEARCH
  - contents      "foo"
  - separator     '/'
```

            is equivalent to

```
History entry with timestamp 1970-01-01T00:00:00:
  = [SEARCH, "foo", '/']
```

            Note: special array syntax for register entries is not recognized here.
2. " = {msgpack-value}": raw values. {msgpack-value} in this case may

have absolutely any type. Special array syntax for register entries is
not recognized here as well.

RUST ft-rust

Since the text for this plugin is rather long it has been put in a separate
file: ft_rust.txt.

SQL ft-sql

Since the text for this plugin is rather long it has been put in a separate
file: ft_sql.txt.

TEX ft-tex-plugin g:tex_flavor

If the first line of a *.tex file has the form

```
%&<format>
```

then this determined the file type: plaintex (for plain TeX), context (for
ConTeXt), or tex (for LaTeX). Otherwise, the file is searched for keywords to
choose context or tex. If no keywords are found, it defaults to plaintex.
You can change the default by defining the variable g:tex_flavor to the format
(not the file type) you use most. Use one of these:

```
let g:tex_flavor = "plain"
let g:tex_flavor = "context"
let g:tex_flavor = "latex"
```

Currently no other formats are recognized.

TYPST ft-typst-plugin

g:typst_recommended_style

By default, the Typst filetype plugin enables the following options:

```
setlocal expandtab shiftwidth=2 softtabstop=2
```

To disable this, set the following variable:

```
let g:typst_recommended_style = 0
```

g:typst_conceal

When TRUE the Typst filetype plugin will set the 'conceallevel' option to 2.

g:typst_folding

When TRUE the Typst filetype plugin will fold headings. (default: FALSE)

To enable:

```
let g:typst_folding = 1
```

g:typst_foldnested

When TRUE the Typst filetype plugin will fold nested heading under their
parents. (default: TRUE)

To disable:

```
let g:typst_foldnested = 0
```

VIM ft-vim-plugin

The Vim filetype plugin defines mappings to move to the start and end of
functions with [[and]]. Move around comments with]" and ["

The mappings can be disabled with:

```
let g:no_vim_maps = 1
```

YAML ft-yaml-plugin

By default, the YAML filetype plugin enables the following options:

```
setlocal shiftwidth=2 softtabstop=2
```

To disable this, set the following variable:

```
let g:yaml_recommended_style = 0
```

ZIG

ft-zig-plugin

g:zig_recommended_style

By default the following indentation options are set, in accordance with Zig's recommended style (https://ziglang.org/documentation/master/):

```
setlocal expandtab shiftwidth=4 softtabstop=4 tabstop=8
```

To disable this behavior, set g:zig_recommended_style to 0:

```
let g:zig_recommended_style = 0
```

g:zig_std_dir

The path to the Zig standard library. The Zig ftplugin reads g:zig_std_dir and appends it to the 'path' for Zig files. Where the Zig standard library is located is system and installation method dependent.

One can automatically set g:zig_std_dir using zig env:

```
let g:zig_std_dir = json_decode(system('zig env'))['std_dir']
```

This can, for example, be put in a FileType :autocmd or user ftplugin to only load when a Zig file is opened.

ZIMBU

ft-zimbu-plugin

The Zimbu filetype plugin defines mappings to move to the start and end of functions with [[and]].

The mappings can be disabled with:

```
let g:no_zimbu_maps = 1
```

Folding folding folds

You can find an introduction on folding in chapter 28 of the user manual.
usr_28.txt

1. Fold methods fold-methods

The folding method can be set with the 'foldmethod' option.

When setting 'foldmethod' to a value other than "manual", all folds are deleted and new ones created. Switching to the "manual" method doesn't remove the existing folds. This can be used to first define the folds automatically and then change them manually.

There are six methods to select folds:

manual	manually define folds
indent	more indent means a higher fold level
expr	specify an expression to define folds
syntax	folds defined by syntax highlighting
diff	folds for unchanged text
marker	folds defined by markers in the text

MANUAL fold-manual

Use commands to manually define the fold regions. This can also be used by a script that parses text to find folds.

The level of a fold is only defined by its nesting. To increase the fold level of a fold for a range of lines, define a fold inside it that has the same lines.

The manual folds are lost when you abandon the file. To save the folds use the :mkview command. The view can be restored later with :loadview.

INDENT fold-indent

The folds are automatically defined by the indent of the lines.

The foldlevel is computed from the indent of the line, divided by the 'shiftwidth' (rounded down). A sequence of lines with the same or higher fold level form a fold, with the lines with a higher level forming a nested fold.

The nesting of folds is limited with 'foldnestmax'.

Some lines are ignored and get the fold level of the line above or below it, whichever is lower. These are empty or white lines and lines starting with a character in 'foldignore'. White space is skipped before checking for characters in 'foldignore'. For C use "#" to ignore preprocessor lines.

When you want to ignore lines in another way, use the "expr" method. The indent() function can be used in 'foldexpr' to get the indent of a line.

EXPR fold-expr

The folds are automatically defined by their foldlevel, like with the "indent" method. The value of the 'foldexpr' option is evaluated to get the foldlevel of a line. Examples:
This will create a fold for all consecutive lines that start with a tab:

```
vim.wo.foldexpr = function()
  return vim.fn.getline(vim.v.lnum):sub(1, 1) == '\t' and 1 or 0
end
```

This will make a fold out of paragraphs separated by blank lines:

```
vim.wo.foldexpr = function()
  if vim.fn.getline(vim.v.lnum):match('^%s*$')
      and vim.fn.getline(vim.v.lnum + 1):match('%S') then
    return '<1'
  end
  return 1
end
```

This does the same:

```
vim.wo.foldexpr = function()
  if vim.fn.getline(vim.v.lnum - 1):match('^%s*$')
      and vim.fn.getline(vim.v.lnum):match('%S') then
    return '>1'
  end
  return 1
end
```

When setting an expression string with ":set" instead, backslashes must be used to escape characters that ":set" handles differently (space, backslash, double quote, etc., see option-backslash):

```
:set foldexpr=getline(v:lnum)[0]==\"\\t\"
```

The most efficient is to use a function value (expr-option-function):

```
vim.wo.foldexpr = MyFoldLevel
```

or to call a Vimscript function without arguments:

```
:set foldexpr=MyFoldLevel()
```

The function must use v:lnum. See expr-option-function.

These are the conditions with which the expression is evaluated:

- The current buffer and window are set for the line.

- The variable "v:lnum" is set to the line number.

The result of foldexpr then determines the fold level as follows:

value	meaning
0	the line is not in a fold
1, 2, ..	the line is in a fold with this level
-1	the fold level is undefined, use the fold level of a line before or after this line, whichever is the lowest.
"="	use fold level from the previous line
"a1", "a2", ..	add one, two, .. to the fold level of the previous line, use the result for the current line
"s1", "s2", ..	subtract one, two, .. from the fold level of the previous line, use the result for the next line
"<1", "<2", ..	a fold with this level ends at this line
">1", ">2", ..	a fold with this level starts at this line

The result values "=", "s" and "a" are more expensive, please see fold-expr-slow.

It is not required to mark the start (end) of a fold with ">1" ("<1"), a fold will also start (end) when the fold level is higher (lower) than the fold level of the previous line.

There must be no side effects from the expression. The text in the buffer, cursor position, the search patterns, options etc. must not be changed. You can change and restore them if you are careful.

If there is some error in the expression, or the resulting value isn't recognized, there is no error message and the fold level will be zero. For debugging the 'debug' option can be set to "msg", the error messages will be visible then.

If the 'foldexpr' expression starts with s: or <SID>, then it is replaced with the script ID (local-function). Examples:

```
set foldexpr=s:MyFoldExpr()
set foldexpr=<SID>SomeFoldExpr()
```

An example of using "a1" and "s1": For a multi-line C comment, a line containing "/*" would return "a1" to start a fold, and a line containing "*/" would return "s1" to end the fold after that line:

```
if match(thisline, '/\*') >= 0
  return 'a1'
elseif match(thisline, '\*/') >= 0
  return 's1'
else
  return '='
endif
```

However, this won't work for single line comments, strings, etc.

foldlevel() can be useful to compute a fold level relative to a previous fold level. But note that foldlevel() may return -1 if the level is not known yet. And it returns the level at the start of the line, while a fold might end in that line.

It may happen that folds are not updated properly. You can use zx or zX to force updating folds.

MINIMIZING COMPUTATIONAL COST fold-expr-slow

Due to its computational cost, this fold method can make Vim unresponsive, especially when the fold level of all lines have to be initially computed. Afterwards, after each change, Vim restricts the computation of foldlevels to those lines whose fold level was affected by it (and reuses the known foldlevels of all the others).

The fold expression should therefore strive to minimize the number of dependent lines needed for the computation of a given line: For example, try to avoid the "=", "a" and "s" return values, because these will require the evaluation of the fold levels on previous lines until an independent fold level is found.

If this proves difficult, the next best thing could be to cache all fold levels in a buffer-local variable (b:foldlevels) that is only updated on b:changedtick:

```
vim.wo.foldexpr = function()
  if vim.b.lasttick == vim.b.changedtick then
    return vim.b.foldlevels[vim.v.lnum]
  end
  vim.b.lasttick = vim.b.changedtick
  local levels = {}
  -- compute foldlevels ...
  vim.b.foldlevels = levels
  return levels[vim.v.lnum]
end
```

A function value also avoids re-parsing the expression for every line (it must still use v:lnum). See expr-option-function.

SYNTAX fold-syntax

A fold is defined by syntax items that have the "fold" argument. :syn-fold

The fold level is defined by nesting folds. The nesting of folds is limited with 'foldnestmax'.

Be careful to specify proper syntax syncing. If this is not done right, folds may differ from the displayed highlighting. This is especially relevant when using patterns that match more than one line. In case of doubt, try using brute-force syncing:

```
:syn sync fromstart
```

DIFF fold-diff

The folds are automatically defined for text that is not part of a change or close to a change.

This method only works properly when the 'diff' option is set for the current window and changes are being displayed. Otherwise the whole buffer will be one big fold.

The 'diffopt' option can be used to specify the context. That is, the number of lines between the fold and a change that are not included in the fold. For example, to use a context of 8 lines:

```
:set diffopt=filler,context:8
```

The default context is six lines.

When 'scrollbind' is also set, Vim will attempt to keep the same folds open in other diff windows, so that the same text is visible.

MARKER fold-marker

Markers in the text tell where folds start and end. This allows you to precisely specify the folds. This will allow deleting and putting a fold, without the risk of including the wrong lines. The 'foldtext' option is normally set such that the text before the marker shows up in the folded line. This makes it possible to give a name to the fold.

Markers can have a level included, or can use matching pairs. Including a level is easier, you don't have to add end markers and avoid problems with non-matching marker pairs. Example:

```
/* global variables {{{1 */
int varA, varB;

/* functions {{{1 */
/* funcA() {{{2 */
void funcA() {}

/* funcB() {{{2 */
void funcB() {}
```

{{{ }}}

A fold starts at a "{{{" marker. The following number specifies the fold level. What happens depends on the difference between the current fold level and the level given by the marker:

1. If a marker with the same fold level is encountered, the previous fold ends and another fold with the same level starts.
2. If a marker with a higher fold level is found, a nested fold is started.
3. If a marker with a lower fold level is found, all folds up to and including this level end and a fold with the specified level starts.

The number indicates the fold level. A zero cannot be used (a marker with level zero is ignored). You can use "}}}" with a digit to indicate the level of the fold that ends. The fold level of the following line will be one less than the indicated level. Note that Vim doesn't look back to the level of the matching marker (that would take too much time). Example:

```
{{{1
fold level here is 1
{{{3
fold level here is 3
}}}3
fold level here is 2
```

You can also use matching pairs of "{{{" and "}}}" markers to define folds.
Each "{{{" increases the fold level by one, each "}}}" decreases the fold
level by one. Be careful to keep the markers matching! Example:

```
{{{
fold level here is 1
{{{
fold level here is 2
}}}
fold level here is 1
```

You can mix using markers with a number and without a number. A useful way of
doing this is to use numbered markers for large folds, and unnumbered markers
locally in a function. For example use level one folds for the sections of
your file like "structure definitions", "local variables" and "functions".
Use level 2 markers for each definition and function, Use unnumbered markers
inside functions. When you make changes in a function to split up folds, you
don't have to renumber the markers.

The markers can be set with the 'foldmarker' option. It is recommended to
keep this at the default value of "{{{,}}}", so that files can be exchanged
between Vim users. Only change it when it is required for the file (e.g., it
contains markers from another folding editor, or the default markers cause
trouble for the language of the file).

fold-create-marker

"zf" can be used to create a fold defined by markers. Vim will insert the
markers for you. Vim will append the start and end marker, as specified with
'foldmarker'. The markers are appended to the end of the line.
'commentstring' is used if it isn't empty.
This does not work properly when:

- The line already contains a marker with a level number. Vim then doesn't know what to do.
- Folds nearby use a level number in their marker which gets in the way.
- The line is inside a comment, 'commentstring' isn't empty and nested comments don't work. For example with C: adding /* {{{ */ inside a comment will truncate the existing comment. Either put the marker before or after the comment, or add the marker manually.

Generally it's not a good idea to let Vim create markers when you already have markers with a level number.

fold-delete-marker

"zd" can be used to delete a fold defined by markers. Vim will delete the
markers for you. Vim will search for the start and end markers, as specified
with 'foldmarker', at the start and end of the fold. When the text around the
marker matches with 'commentstring', that text is deleted as well.
This does not work properly when:

- A line contains more than one marker and one of them specifies a level. Only the first one is removed, without checking if this will have the desired effect of deleting the fold.
- The marker contains a level number and is used to start or end several folds at the same time.

2. Fold commands

fold-commands E490

All folding commands start with "z". Hint: the "z" looks like a folded piece
of paper, if you look at it from the side.

CREATING AND DELETING FOLDS

zf E350

zf{motion} or
{Visual}zf   Operator to create a fold.
                This only works when 'foldmethod' is "manual" or "marker".
                The new fold will be closed for the "manual" method.
                'foldenable' will be set.
                Also see fold-create-marker.

zF

zF   Create a fold for [count] lines. Works like "zf".

:{range}fo[ld] :fold :fo
Create a fold for the lines in {range}. Works like "zf".

zd E351
zd Delete one fold at the cursor. When the cursor is on a folded line, that fold is deleted. Nested folds are moved one level up. In Visual mode one level of all folds (partially) in the selected area are deleted.
Careful: This easily deletes more folds than you expect and there is no undo for manual folding.
This only works when 'foldmethod' is "manual" or "marker".
Also see fold-delete-marker.

zD
zD Delete folds recursively at the cursor. In Visual mode all folds (partially) in the selected area and all nested folds in them are deleted.
This only works when 'foldmethod' is "manual" or "marker".
Also see fold-delete-marker.

zE E352
zE Eliminate all folds in the window.
This only works when 'foldmethod' is "manual" or "marker".
Also see fold-delete-marker.

OPENING AND CLOSING FOLDS

A fold smaller than 'foldminlines' will always be displayed like it was open.
Therefore the commands below may work differently on small folds.

zo
zo Open one fold under the cursor. When a count is given, that many folds deep will be opened. In Visual mode one level of folds is opened for all lines in the selected area.

zO
zO Open all folds under the cursor recursively. Folds that don't contain the cursor line are unchanged.
In Visual mode it opens all folds that are in the selected area, also those that are only partly selected.

zc
zc Close one fold under the cursor. When a count is given, that many folds deep are closed. In Visual mode one level of folds is closed for all lines in the selected area.
'foldenable' will be set.

zC
zC Close all folds under the cursor recursively. Folds that don't contain the cursor line are unchanged.
In Visual mode it closes all folds that are in the selected area, also those that are only partly selected.
'foldenable' will be set.

za
za Summary: Toggle the fold under the cursor.
When on a closed fold: open it. When folds are nested, you may have to use "za" several times. When a count is given, that many closed folds are opened.
When on an open fold: close it and set 'foldenable'. This will only close one level, since using "za" again will open the fold. When a count is given that many folds will be closed (that's not the same as repeating "za" that many times).

zA
zA When on a closed fold: open it recursively.
When on an open fold: close it recursively and set 'foldenable'.

zv
zv View cursor line: Open just enough folds to make the line in which the cursor is located not folded.

zx
zx Update folds: Undo manually opened and closed folds: re-apply 'foldlevel', then do "zv": View cursor line.
Also forces recomputing folds. This is useful when using 'foldexpr' and the buffer is changed in a way that results in folds not to be updated properly.

zX
zX Undo manually opened and closed folds: re-apply 'foldlevel'.
Also forces recomputing folds, like zx.

zm
zm Fold more: Subtract v:count1 from 'foldlevel'. If

'foldlevel' was already zero nothing happens.
'foldenable' will be set.

zM

zM Close all folds: set 'foldlevel' to 0.
'foldenable' will be set.

zr

zr Reduce folding: Add v:count1 to 'foldlevel'.

zR

zR Open all folds. This sets 'foldlevel' to highest fold level.

:foldo :foldopen

:{range}foldo[pen][!]
Open folds in {range}. When [!] is added all folds are
opened. Useful to see all the text in {range}. Without [!]
one level of folds is opened.

:foldc :foldclose

:{range}foldc[lose][!]
Close folds in {range}. When [!] is added all folds are
closed. Useful to hide all the text in {range}. Without [!]
one level of folds is closed.

zn

zn Fold none: reset 'foldenable'. All folds will be open.

zN

zN Fold normal: set 'foldenable'. All folds will be as they
were before.

zi

zi Invert 'foldenable'.

MOVING OVER FOLDS

[z

[z Move to the start of the current open fold. If already at the
start, move to the start of the fold that contains it. If
there is no containing fold, the command fails.
When a count is used, repeats the command [count] times.

]z

]z Move to the end of the current open fold. If already at the
end, move to the end of the fold that contains it. If there
is no containing fold, the command fails.
When a count is used, repeats the command [count] times.

zj

zj Move downwards to the start of the next fold. A closed fold
is counted as one fold.
When a count is used, repeats the command [count] times.
This command can be used after an operator.

zk

zk Move upwards to the end of the previous fold. A closed fold
is counted as one fold.
When a count is used, repeats the command [count] times.
This command can be used after an operator.

EXECUTING COMMANDS ON FOLDS

:[range]foldd[oopen] {cmd} :foldd :folddo :folddoopen
Execute {cmd} on all lines that are not in a closed fold.
When [range] is given, only these lines are used.
Each time {cmd} is executed the cursor is positioned on the
line it is executed for.
This works like the ":global" command: First all lines that
are not in a closed fold are marked. Then the {cmd} is
executed for all marked lines. Thus when {cmd} changes the
folds, this has no influence on where it is executed (except
when lines are deleted, of course).
Example:

```
:folddoopen s/end/loop_end/ge
```

Note the use of the "e" flag to avoid getting an error message
where "end" doesn't match.

:[range]folddoc[losed] {cmd} :folddoc :folddoclosed
Execute {cmd} on all lines that are in a closed fold.
Otherwise like ":folddoopen".

3. Fold options

fold-options

COLORS

fold-colors

The colors of a closed fold are set with the Folded group hl-Folded. The
colors of the fold column are set with the FoldColumn group hl-FoldColumn.
Example to set the colors:

```
:highlight Folded guibg=grey guifg=blue
:highlight FoldColumn guibg=darkgrey guifg=white
```

FOLDLEVEL

fold-foldlevel

'foldlevel' is a number option: The higher the more folded regions are open.
When 'foldlevel' is 0, all folds are closed.
When 'foldlevel' is positive, some folds are closed.
When 'foldlevel' is very high, all folds are open.
'foldlevel' is applied when it is changed. After that manually folds can be
opened and closed.
When increased, folds above the new level are opened. No manually opened
folds will be closed.
When decreased, folds above the new level are closed. No manually closed
folds will be opened.

FOLDTEXT

fold-foldtext

'foldtext' is a string option that specifies an expression. This expression
is evaluated to obtain the text displayed for a closed fold. Example:

```
vim.wo.foldtext = function()
  local line = vim.fn.getline(vim.v.foldstart)
  return vim.v.folddashes .. line:gsub('/%*', ''):gsub('%*/', '')
end
```

This shows the first line of the fold, with "/*" and "*/" removed.
Note the use of backslashes to avoid some characters to be interpreted by the
":set" command. It is much simpler to define a function and call it:

```
:set foldtext=MyFoldText()
:function MyFoldText()
:  let line = getline(v:foldstart)
:  let sub = substitute(line, '/\*\|\*/\|{{{\d\=', '', 'g')
:  return v:folddashes .. sub
:endfunction
```

Evaluating 'foldtext' is done in the sandbox. The current window is set to
the window that displays the line.

Errors are ignored. For debugging set the 'debug' option to "throw".

The default value is foldtext(). This returns a reasonable text for most
types of folding. If you don't like it, you can specify your own 'foldtext'
expression. It can use these special Vim variables:

v:foldstart	line number of first line in the fold
v:foldend	line number of last line in the fold
v:folddashes	a string that contains dashes to represent the foldlevel.
v:foldlevel	the foldlevel of the fold

If the result is a List, it is parsed and drawn like "overlay" virtual text
(see nvim_buf_set_extmark()), otherwise the result is converted to a string
where a TAB is replaced with a space and unprintable characters are made into
printable characters.

The resulting line is truncated to fit in the window, it never wraps.
When there is room after the text, it is filled with the character specified
by 'fillchars'.

If the 'foldtext' expression starts with s: or <SID>, then it is replaced
with the script ID (local-function). Examples:

```
set foldtext=s:MyFoldText()
set foldtext=<SID>SomeFoldText()
```

Note that backslashes need to be used for characters that the ":set" command
handles differently: Space, backslash and double-quote. option-backslash

FOLDCOLUMN

fold-foldcolumn

'foldcolumn' is a number, which sets the width for a column on the side of the
window to indicate folds. When it is zero, there is no foldcolumn. A normal
value is auto:9. The maximum is 9.

An open fold is indicated with a column that has a '-' at the top and '|'
characters below it. This column stops where the open fold stops. When folds
nest, the nested fold is one character right of the fold it's contained in.

A closed fold is indicated with a '+'.

These characters can be changed with the 'fillchars' option.

Where the fold column is too narrow to display all nested folds, digits are shown to indicate the nesting level. To override this behavior you can use the "foldinner" character of the 'fillchars' option.

The mouse can also be used to open and close folds by clicking in the fold column:

- Click on a '+' to open the closed fold at this row.
- Click on any other non-blank character to close the open fold at this row.

OTHER OPTIONS

'foldenable'	'fen':	Open all folds while not set.
'foldexpr'	'fde':	Expression used for "expr" folding.
'foldignore'	'fdi':	Characters used for "indent" folding.
'foldmarker'	'fmr':	Defined markers used for "marker" folding.
'foldmethod'	'fdm':	Name of the current folding method.
'foldminlines'	'fml':	Minimum number of screen lines for a fold to be displayed closed.
'foldnestmax'	'fdn':	Maximum nesting for "indent" and "syntax" folding.
'foldopen'	'fdo':	Which kinds of commands open closed folds.
'foldclose'	'fcl':	When the folds not under the cursor are closed.

4. Behavior of folds fold-behavior

When moving the cursor upwards or downwards and when scrolling, the cursor will move to the first line of a sequence of folded lines. When the cursor is already on a folded line, it moves to the next unfolded line or the next closed fold.

While the cursor is on folded lines, the cursor is always displayed in the first column. The ruler does show the actual cursor position, but since the line is folded, it cannot be displayed there.

Many movement commands handle a sequence of folded lines like an empty line. For example, the "w" command stops once in the first column.

When starting a search in a closed fold it will not find a match in the current fold. It's like a forward search always starts from the end of the closed fold, while a backwards search starts from the start of the closed fold.

When in Insert mode, the cursor line is never folded. That allows you to see what you type!

When using an operator, a closed fold is included as a whole. Thus "dl" deletes the whole closed fold under the cursor.

For Ex commands that operate on buffer lines, the range is adjusted to always start at the first line of a closed fold and end at the last line of a closed fold. Thus, this command:

```
:s/foo/bar/g
```

when used with the cursor on a closed fold, will replace "foo" with "bar" in all lines of the fold.
This does not happen for :folddoopen and :folddoclosed.

Note that for some Ex commands like :source the range is only adjusted when using a two line specifiers [range].

When editing a buffer that has been edited before, the last used folding settings are used again. For manual folding the defined folds are restored. For all folding methods the manually opened and closed folds are restored. If this buffer has been edited in this window, the values from back then are used. Otherwise the values from the window where the buffer was edited last are used.

ADA FILE TYPE PLUG-INS REFERENCE MANUAL~

ADA

1. Syntax Highlighting ~

This mode is designed for the 2005 edition of Ada ("Ada 2005"), which includes support for objected-programming, protected types, and so on. It handles code written for the original Ada language ("Ada83", "Ada87", "Ada95") as well, though code which uses Ada 2005-only keywords will be wrongly colored (such code should be fixed anyway). For more information about Ada, see http://www.adapower.com.

The Ada mode handles a number of situations cleanly.

For example, it knows that the "-" in "-5" is a number, but the same character in "A-5" is an operator. Normally, a "with" or "use" clause referencing another compilation unit is coloured the same way as C's "#include" is coloured. If you have "Conditional" or "Repeat" groups coloured differently, then "end if" and "end loop" will be coloured as part of those respective groups.

You can set these to different colours using vim's "highlight" command (e.g., to change how loops are displayed, enter the command ":hi Repeat" followed by the colour specification; on simple terminals the colour specification ctermfg=White often shows well).

There are several options you can select in this Ada mode. See ft-ada-options for a complete list.

To enable them, assign a value to the option. For example, to turn one on:

```
let g:ada_standard_types = 1
```

To disable them use ":unlet". Example:

```
unlet g:ada_standard_types
```

You can just use ":" and type these into the command line to set these temporarily before loading an Ada file. You can make these option settings permanent by adding the "let" command(s), without a colon, to your init.vim file.

Even on a slow (90Mhz) PC this mode works quickly, but if you find the performance unacceptable, turn on g:ada_withuse_ordinary.

Syntax folding instructions (fold-syntax) are added when g:ada_folding is set.

2. File type Plug-in ~

ft-ada-indent ft-ada-plugin

The Ada plug-in provides support for:

- auto indenting (indent.txt)
- insert completion (i_CTRL-N)
- user completion (i_CTRL-X_CTRL-U)
- tag searches (tagsrch.txt)
- Quick Fix (quickfix.txt)
- backspace handling ('backspace')
- comment handling ('comments', 'commentstring')

The plug-in only activates the features of the Ada mode whenever an Ada file is opened and adds Ada related entries to the main and pop-up menu.

3. Omni Completion ~

ft-ada-omni

The Ada omni-completions (i_CTRL-X_CTRL-O) uses tags database created either by "gnat xref -v" or the "Universal Ctags" (https://ctags.io). The complete function will automatically detect which tool was used to create the tags file.

3.1 Omni Completion with "gnat xref" ~

gnat-xref

GNAT XREF uses the compiler internal information (ali-files) to produce the tags file. This has the advantage to be 100% correct and the option of deep nested analysis. However the code must compile, the generator is quite

slow and the created tags file contains only the basic Ctags information for each entry - not enough for some of the more advanced Vim code browser plug-ins.

NOTE: "gnat xref -v" is very tricky to use as it has almost no diagnostic output - If nothing is printed then usually the parameters are wrong. Here some important tips:

1) You need to compile your code first and use the "-aO" option to point to your .ali files.
2) "gnat xref -v ../Include/adacl.ads" won't work - use the "gnat xref -v -aI../Include adacl.ads" instead.
3) gnat xref -v -aI../Include *.ad? won't work - use "cd ../Include" and then gnat xref -v *.ad?
4) Project manager support is completely broken - don't even try "gnat xref -Padacl.gpr".
5) Vim is faster when the tags file is sorted - use "sort --unique --ignore-case --output=tags tags" .
6) Remember to insert "!_TAG_FILE_SORTED 2 %sort ui" as first line to mark the file assorted.

3.2 Omni Completion with "ctags"~

ada-ctags

Universal/Exuberant Ctags use their own multi-language code parser. The parser is quite fast, produces a lot of extra information and can run on files which currently do not compile.

There are also lots of other Vim-tools which use Universal/Exuberant Ctags. Universal Ctags is preferred, Exuberant Ctags is no longer being developed.

You will need to install Universal Ctags which is available from https://ctags.io

The Ada parser for Universal/Exuberant Ctags is fairly new - don't expect complete support yet.

4. Compiler Support ~

ada-compiler

The Ada mode supports more than one Ada compiler and will automatically load the compiler set in g:ada_default_compiler whenever an Ada source is opened. The provided compiler plug-ins are split into the actual compiler plug-in and a collection of support functions and variables. This allows the easy development of specialized compiler plug-ins fine tuned to your development environment.

4.1 GNAT ~

compiler-gnat

GNAT is the only free (beer and speech) Ada compiler available. There are several versions available which differ in the licence terms used.

The GNAT compiler plug-in will perform a compile on pressing <F7> and then immediately shows the result. You can set the project file to be used by setting:

```
call g:gnat.Set_Project_File ('my_project.gpr')
```

Project file names containing spaces or shell special characters are supported.

Setting a project file will also create a Vim session (views-sessions) so - like with the GPS - opened files, window positions etc. will be remembered separately for all projects.

gnat_members

GNAT OBJECT

g:gnat.Make()

g:gnat.Make()
Calls g:gnat.Make_Command and displays the result inside a quickfix window.

g:gnat.Pretty()

g:gnat.Pretty()
Calls g:gnat.Pretty_Program

g:gnat.Find()

g:gnat.Find()
Calls g:gnat.Find_Program

g:gnat.Tags()

g:gnat.Tags()
Calls g:gnat.Tags_Command

g:gnat.Set_Project_File()

g:gnat.Set_Project_File([{file}])
Set gnat project file and load associated session. An open project will be closed and the session written. If called

without file name the file selector opens for selection of a
project file. If called with an empty string then the project
and associated session are closed.

g:gnat.Project_File

g:gnat.Project_File   string
Current project file.

g:gnat.Make_Command

g:gnat.Make_Command   string
External command used for g:gnat.Make() ('makeprg').

g:gnat.Pretty_Program

g:gnat.Pretty_Program   string
External command used for g:gnat.Pretty()

g:gnat.Find_Program

g:gnat.Find_Program   string
External command used for g:gnat.Find()

g:gnat.Tags_Command

g:gnat.Tags_Command   string
External command used for g:gnat.Tags()

g:gnat.Error_Format

g:gnat.Error_Format   string
Error format ('errorformat')

4.2 Dec Ada ~

compiler-hpada compiler-decada
compiler-vaxada compiler-compaqada

Dec Ada (also known by - in chronological order - VAX Ada, Dec Ada, Compaq Ada
and HP Ada) is a fairly dated Ada 83 compiler. Support is basic: <F7> will
compile the current unit.

The Dec Ada compiler expects the package name and not the file name to be
passed as a parameter. The compiler plug-in supports the usual file name
convention to convert the file into a unit name. Both '-' and '__' are allowed
as separators.

decada_members

DEC ADA OBJECT

g:decada.Make()

g:decada.Make()   function
Calls g:decada.Make_Command and displays the result inside a
quickfix window.

g:decada.Unit_Name()

g:decada.Unit_Name()   function
Get the Unit name for the current file.

g:decada.Make_Command

g:decada.Make_Command   string
External command used for g:decada.Make() ('makeprg').

g:decada.Error_Format

g:decada.Error_Format   string
Error format ('errorformat').

5. References ~

ada-reference

5.1 Options ~

ft-ada-options

g:ada_standard_types

g:ada_standard_types   bool (true when exists)
Highlight types in package Standard (e.g., "Float").

g:ada_space_errors

g:ada_no_trail_space_error

g:ada_no_tab_space_error

g:ada_all_tab_usage

g:ada_space_errors   bool (true when exists)
Highlight extraneous errors in spaces ...
g:ada_no_trail_space_error

- but ignore trailing spaces at the end of a line

g:ada_no_tab_space_error
- but ignore tabs after spaces

g:ada_all_tab_usage
- highlight all tab use

g:ada_line_errors

g:ada_line_errors bool (true when exists)
Highlight lines which are too long. **Note:** This highlighting
option is quite CPU intensive.

g:ada_rainbow_color

g:ada_rainbow_color bool (true when exists)
Use rainbow colours for '(' and ')'. You need the
rainbow_parenthesis for this to work.

g:ada_folding

g:ada_folding set ("sigpft")
Use folding for Ada sources.
's': activate syntax folding on load
'p': fold packages
'f': fold functions and procedures
't': fold types
'c': fold conditionals
'g': activate gnat pretty print folding on load
'i': lone "is" folded with line above
'b': lone "begin" folded with line above
'p': lone "private" folded with line above
'x': lone "exception" folded with line above
'i': activate indent folding on load

Note: Syntax folding is in an early (unusable) stage and
indent or gnat pretty folding is suggested.

For gnat pretty folding to work the following settings are
suggested: -cl3 -M79 -c2 -c3 -c4 -A1 -A2 -A3 -A4 -A5

For indent folding to work the following settings are
suggested: shiftwidth=3 softtabstop=3

g:ada_abbrev

g:ada_abbrev bool (true when exists)
Add some abbreviations. This feature is more or less superseded
by the various completion methods.

g:ada_withuse_ordinary

g:ada_withuse_ordinary bool (true when exists)
Show "with" and "use" as ordinary keywords (when used to
reference other compilation units they're normally highlighted
specially).

g:ada_begin_preproc

g:ada_begin_preproc bool (true when exists)
Show all begin-like keywords using the colouring of C
preprocessor commands.

g:ada_omni_with_keywords

g:ada_omni_with_keywords
Add Keywords, Pragmas, Attributes to omni-completions
(compl-omni). **Note:** You can always complete then with user
completion (i_CTRL-X_CTRL-U).

g:ada_extended_tagging

g:ada_extended_tagging enum ("jump", "list")
use extended tagging, two options are available
"jump": use tjump to jump.
"list": add tags quick fix list.
Normal tagging does not support function or operator
overloading as these features are not available in C and
tagging was originally developed for C.

g:ada_extended_completion

g:ada_extended_completion
Uses extended completion for <C-N> and <C-R> completions
(i_CTRL-N). In this mode the '.' is used as part of the
identifier so that 'Object.Method' or 'Package.Procedure' are
completed together.

g:ada_gnat_extensions

g:ada_gnat_extensions bool (true when exists)
Support GNAT extensions.

g:ada_with_gnat_project_files

g:ada_with_gnat_project_files   bool (true when exists)
                Add gnat project file keywords and Attributes.

g:ada_default_compiler

g:ada_default_compiler   string
                set default compiler. Currently supported are "gnat" and
                "decada".

An "exists" type is a boolean considered true when the variable is defined and
false when the variable is undefined. The value to which the variable is set
makes no difference.

5.2 Commands ~

ft-ada-commands

:AdaRainbow :AdaRainbow

                Toggles rainbow colour (g:ada_rainbow_color) mode for
                '(' and ')'.

:AdaLines :AdaLines

                Toggles line error (g:ada_line_errors) display.

:AdaSpaces :AdaSpaces

                Toggles space error (g:ada_space_errors) display.

:AdaTagDir :AdaTagDir

                Creates tags file for the directory of the current file.

:AdaTagFile :AdaTagFile

                Creates tags file for the current file.

:AdaTypes :AdaTypes

                Toggles standard types (g:ada_standard_types) colour.

:GnatFind :GnatFind

                Calls g:gnat.Find()

:GnatPretty :GnatPretty

                Calls g:gnat.Pretty()

:GnatTags :GnatTags

                Calls g:gnat.Tags()

5.3 Variables ~

ft-ada-variables

g:gnat

g:gnat   object
                Control object which manages GNAT compiles. The object
                is created when the first Ada source code is loaded provided
                that g:ada_default_compiler is set to "gnat". See
                gnat_members for details.

g:decada

g:decada   object
                Control object which manages Dec Ada compiles. The object
                is created when the first Ada source code is loaded provided
                that g:ada_default_compiler is set to "decada". See
                decada_members for details.

5.4 Constants ~

ft-ada-constants

All constants are locked. See :lockvar for details.

g:ada#WordRegex

g:ada#WordRegex   string
                Regular expression to search for Ada words.

g:ada#DotWordRegex

g:ada#DotWordRegex   string
                Regular expression to search for Ada words separated by dots.

g:ada#Comment

g:ada#Comment   string
                Regular expression to search for Ada comments.

g:ada#Keywords

g:ada#Keywords   list of dictionaries

List of keywords, attributes etc. pp. in the format used by
omni completion. See complete-items for details.

g:ada#Ctags_Kinds

g:ada#Ctags_Kinds dictionary of lists
Dictionary of the various kinds of items which the Ada support
for Ctags generates.

5.5 Functions ~

ft-ada-functions

ada#Word([{line}, {col}]) ada#Word()

Return full name of Ada entity under the cursor (or at given
line/column), stripping white space/newlines as necessary.

ada#List_Tag([{line}, {col}]) ada#Listtags()

List all occurrences of the Ada entity under the cursor (or at
given line/column) inside the quick-fix window.

ada#Jump_Tag ({ident}, {mode}) ada#Jump_Tag()

List all occurrences of the Ada entity under the cursor (or at
given line/column) in the tag jump list. Mode can either be
"tjump" or "stjump".

ada#Create_Tags ({option}) ada#Create_Tags()

Creates tag file using Ctags. The option can either be "file"
for the current file, "dir" for the directory of the current
file or a file name.

gnat#Insert_Tags_Header() gnat#Insert_Tags_Header()

Adds the tag file header (!_TAG_) information to the current
file which are missing from the GNAT XREF output.

ada#Switch_Syntax_Option ({option}) ada#Switch_Syntax_Option()

Toggles highlighting options on or off. Used for the Ada menu.

gnat#New()

gnat#New ()
Create a new gnat object. See g:gnat for details.

6. Extra Plugins ~

ada-extra-plugins

You can optionally install the following extra plug-ins. They work well with
Ada and enhance the ability of the Ada mode:

backup.vim
https://www.vim.org/scripts/script.php?script_id=1537
Keeps as many backups as you like so you don't have to.

rainbow_parenthesis.vim
https://www.vim.org/scripts/script.php?script_id=1561
Very helpful since Ada uses only '(' and ')'.

nerd_comments.vim
https://www.vim.org/scripts/script.php?script_id=1218
Excellent commenting and uncommenting support for almost any
programming language.

matchit.vim
https://www.vim.org/scripts/script.php?script_id=39
'%' jumping for any language. The normal '%' jump only works for '{}'
style languages. The Ada mode will set the needed search patterns.

taglist.vim
https://www.vim.org/scripts/script.php?script_id=273
Source code explorer sidebar. There is a patch for Ada available.

The GNU Ada Project distribution (https://sourceforge.net/projects/gnuada/) of Vim
contains all of the above.

CONTENTS

1. Introduction hare-intro

2. Filetype plugin hare-plugin

3. Settings hare-settings

INTRODUCTION hare-intro

This plugin provides syntax highlighting, indentation, and other functionality for the Hare programming language. Support is also provided for README files inside Hare modules, but this must be enabled by setting g:filetype_haredoc.

FILETYPE PLUGIN hare-plugin

This plugin automatically sets the value of 'path' to include the contents of the HAREPATH environment variable, allowing commands such as gf to directly open standard library or third-party modules. If HAREPATH is not set, it defaults to the recommended paths for most Unix-like filesystems, namely /usr/src/hare/stdlib and /usr/src/hare/third-party.

SETTINGS hare-settings

This plugin provides a small number of variables that you can define in your vimrc to configure its behavior.

g:filetype_haredoc

This plugin is able to automatically detect Hare modules and set the "haredoc" filetype for any README files. As the recursive directory search used as a heuristic has a minor performance impact, this feature is disabled by default and must be specifically opted into:

```
let g:filetype_haredoc = 1
```

See g:haredoc_search_depth for ways to tweak the searching behavior.

g:hare_recommended_style

The following options are set by default, in accordance with the official Hare style guide:

```
setlocal noexpandtab
setlocal shiftwidth=0
setlocal softtabstop=0
setlocal tabstop=8
setlocal textwidth=80
```

To disable this behavior:

```
let g:hare_recommended_style = 0
```

g:hare_space_error

By default, trailing whitespace and tabs preceded by space characters are highlighted as errors. This is automatically turned off when in insert mode. To disable this highlighting completely:

```
let g:hare_space_error = 0
```

g:haredoc_search_depth

By default, when g:filetype_haredoc is enabled, only the current directory and its immediate subdirectories are searched for Hare files. The maximum search depth may be adjusted with:

```
let g:haredoc_search_depth = 2
```

Value	Effect~
0	Only search the current directory.
1	Search the current directory and immediate subdirectories.
2	Search the current directory and two levels of subdirectories.

The maximum search depth can be set to any integer, but using values higher than 2 is not recommended, and will likely provide no tangible benefit in most situations.

Author: Peter Provost <https://www.github.com/PProvost>
License: Apache 2.0
URL: https://github.com/PProvost/vim-ps1

INTRODUCTION

This plugin provides Vim syntax, indent and filetype detection for Windows PowerShell scripts, modules, and XML configuration files.

ABOUT ps1-about

Grab the latest version or report a bug on GitHub:

https://github.com/PProvost/vim-ps1

FOLDING ps1-folding

The ps1 syntax file provides syntax folding (see :syn-fold) for script blocks and digital signatures in scripts.

When 'foldmethod' is set to "syntax" then function script blocks will be folded unless you use the following in your .vimrc or before opening a script:

```
:let g:ps1_nofold_blocks = 1
```

Digital signatures in scripts will also be folded unless you use:

```
:let g:ps1_nofold_sig = 1
```

Note: syntax folding might slow down syntax highlighting significantly, especially for large files.

COMPILER ps1-compiler

The powershell :compiler script configures :make to execute the script in PowerShell.

It tries to pick a smart default PowerShell command: pwsh if available and powershell otherwise, but you can customize the command:

```
:let g:ps1_makeprg_cmd = '/path/to/pwsh'
```

To configure whether to show the exception type information:

```
:let g:ps1_efm_show_error_categories = 1
```

KEYWORD LOOKUP ps1-keyword

To look up keywords using PowerShell's Get-Help, press the K key. For more convenient paging, the pager less should be installed, which is included in many Linux distributions and in macOS.

Many other distributions are available for Windows like https://chocolatey.org/packages/less/. Make sure less is in a directory listed in the PATH environment variable, which chocolatey above does.

Vim-raku provides syntax highlighting, indentation, and other support for
editing Raku programs.

1. Using Unicode in your Raku files raku-unicode

1. Using Unicode in your Raku files raku-unicode

Defining new operators using Unicode symbols is a good way to make your
Raku program easy to read. See:
https://perl6advent.wordpress.com/2012/12/18/day-18-formulas-resistance-is-futile/

While Raku does define ASCII alternatives for some common operators (see
https://docs.raku.org/language/unicode_ascii), using the full range of
Unicode operators is highly desirable. Your operating system provides input
facilities, but using the features built in to Vim may be preferable.

The natural way to produce these symbols in Vim is to use digraph shortcuts
(:help digraphs-use). Many of them are defined; type :digraphs to get
the list. A convenient way to read the list of digraphs is to save them in a
file. From the shell:

```
vim +'redir >/tmp/vim-digraphs-listing.txt' +digraphs +'redir END' +q
```

Some of them are available with standard Vim digraphs:

```
<< «    /0 ∅    !< ≮
>> »    Ob ∘    !> ≯
., …    00 ∞    (C ⊂
(U ∩    -: ÷    )C ⊃
)U ∪    (_ ⊆    >= ≥
?= ≅    )_ ⊇    =< ≤
(- ∈    ?= ≅    != ≠
-) ∋    ?- ≃
```

The Greek alphabet is available with "*" followed by a similar Latin symbol:

```
*p π
*t τ
*X ×
```

Numbers, subscripts and superscripts are available with 's' and 'S':

```
0s ₀    0S ⁰
1s ₁    1S ¹
2s ₂    9S ⁹
```

But some don't come defined by default. Those are digraph definitions you can
add in your ~/.vimrc file.

```
exec 'digraph \\ ' .. char2nr('\')
exec 'digraph \< ' .. char2nr('⪇')
exec 'digraph \> ' .. char2nr('⪈')
exec 'digraph (L ' .. char2nr('⫋')
exec 'digraph )L ' .. char2nr('⫌')
exec 'digraph (/ ' .. char2nr('⊄')
exec 'digraph )/ ' .. char2nr('⊅')
exec 'digraph )/ ' .. char2nr('⊅')
exec 'digraph U+ ' .. char2nr('ʉ')
exec 'digraph 0- ' .. char2nr('ɵ')
" Euler's constant
exec 'digraph ne ' .. char2nr('ℯ')
" Raku's atomic operations marker
exec 'digraph @@ ' .. char2nr('⚛')
```

Alternatively, you can write Insert mode abbreviations that convert ASCII-
based operators into their single-character Unicode equivalent.

```
iabbrev <buffer> !(<) ⊄
iabbrev <buffer> !(<=) ⊈
iabbrev <buffer> !(>) ⊅
iabbrev <buffer> !(>=) ⊉
iabbrev <buffer> !(cont) ∌
iabbrev <buffer> !(elem) ∉
iabbrev <buffer> != ≠
iabbrev <buffer> (&) ∩
iabbrev <buffer> (+) ⊎
iabbrev <buffer> (-) \
iabbrev <buffer> (.) ⊎
iabbrev <buffer> (<) ⊂
iabbrev <buffer> (<+) ⩽
iabbrev <buffer> (<=) ⊆
iabbrev <buffer> (>) ⊃
iabbrev <buffer> (>+) ⩾
iabbrev <buffer> (>=) ⊇
iabbrev <buffer> (\|) ∪
iabbrev <buffer> (^) ⊖
iabbrev <buffer> (atomic) ⚛
iabbrev <buffer> (cont) ∋
iabbrev <buffer> (elem) ∈
iabbrev <buffer> * ×
iabbrev <buffer> **0 ⁰
iabbrev <buffer> **1 ¹
iabbrev <buffer> **2 ²
iabbrev <buffer> **3 ³
iabbrev <buffer> **4 ⁴
iabbrev <buffer> **5 ⁵
iabbrev <buffer> **6 ⁶
iabbrev <buffer> **7 ⁷
iabbrev <buffer> **8 ⁸
iabbrev <buffer> **9 ⁹
iabbrev <buffer> ... …
iabbrev <buffer> / ÷
iabbrev <buffer> << «
iabbrev <buffer> <<[=]<< «=«
iabbrev <buffer> <<[=]>> «=»
iabbrev <buffer> <= ≤
iabbrev <buffer> =~= ≅
iabbrev <buffer> >= ≥
iabbrev <buffer> >> »
iabbrev <buffer> >>[=]<< »=«
iabbrev <buffer> >>[=]>> »=»
iabbrev <buffer> Inf ∞
iabbrev <buffer> atomic-add-fetch ⚛+=
iabbrev <buffer> atomic-assign ⚛=
iabbrev <buffer> atomic-fetch ⚛
iabbrev <buffer> atomic-dec-fetch --⚛
iabbrev <buffer> atomic-fetch-dec ⚛--
iabbrev <buffer> atomic-fetch-inc ⚛++
iabbrev <buffer> atomic-inc-fetch ++⚛
iabbrev <buffer> atomic-sub-fetch ⚛-=
iabbrev <buffer> e 𝑒
iabbrev <buffer> o ∘
iabbrev <buffer> pi π
iabbrev <buffer> set() ∅
iabbrev <buffer> tau τ
```

CONTENTS

1. Introduction rust-intro
2. Settings rust-settings
3. Commands rust-commands
4. Mappings rust-mappings

INTRODUCTION rust-intro

This plugin provides syntax and supporting functionality for the Rust filetype.

SETTINGS rust-settings

This plugin has a few variables you can define in your vimrc that change the behavior of the plugin.

Some variables can be set buffer local (b: prefix), and the buffer local will take precedence over the global g: counterpart.

g:rustc_path

g:rustc_path
Set this option to the path to rustc for use in the :RustRun and :RustExpand commands. If unset, "rustc" will be located in $PATH:

```
let g:rustc_path = $HOME."/bin/rustc"
```

g:rustc_makeprg_no_percent

g:rustc_makeprg_no_percent
Set this option to 1 to have 'makeprg' default to "rustc" instead of "rustc %":

```
let g:rustc_makeprg_no_percent = 1
```

g:rust_conceal

g:rust_conceal
Set this option to turn on the basic conceal support:

```
let g:rust_conceal = 1
```

g:rust_conceal_mod_path

g:rust_conceal_mod_path
Set this option to turn on conceal for the path connecting token "::":

```
let g:rust_conceal_mod_path = 1
```

g:rust_conceal_pub

g:rust_conceal_pub
Set this option to turn on conceal for the "pub" token:

```
let g:rust_conceal_pub = 1
```

g:rust_recommended_style

g:rust_recommended_style
Set this option to enable vim indentation and textwidth settings to conform to style conventions of the Rust style guide (i.e. use 4 spaces for indents and set 'textwidth' to 100). This option is enabled by default. To disable it:

```
let g:rust_recommended_style = 0
```

g:rust_fold

g:rust_fold
Set this option to turn on folding:

```
let g:rust_fold = 1
```

Value	Effect
0	No folding
1	Braced blocks are folded. All folds are open by default.
2	Braced blocks are folded. 'foldlevel' is left at the global value (all folds are closed by default).

g:rust_bang_comment_leader

g:rust_bang_comment_leader
Set this option to 1 to preserve the leader on multi-line doc comments
using the /*! syntax:

```
let g:rust_bang_comment_leader = 1
```

g:rust_use_custom_ctags_defs

g:rust_use_custom_ctags_defs
Set this option to 1 if you have customized ctags definitions for Rust
and do not wish for those included with rust.vim to be used:

```
let g:rust_use_custom_ctags_defs = 1
```

NOTE: rust.vim's built-in definitions are only used for the Tagbar Vim
plugin, if you have it installed, AND if Universal Ctags is not
detected. This is because Universal Ctags already has built-in
support for Rust when used with Tagbar.

Also, note that when using ctags other than Universal Ctags, it is not
automatically used when generating tags files that Vim can use to
navigate to definitions across different source files. Feel free to
copy rust.vim/ctags/rust.ctags into your own ~/.ctags if you wish
to generate tags files.

g:ftplugin_rust_source_path

g:ftplugin_rust_source_path
Set this option to a path that should be prepended to 'path' for Rust
source files:

```
let g:ftplugin_rust_source_path = $HOME . '/dev/rust'
```

g:rustfmt_command

g:rustfmt_command
Set this option to the name of the "rustfmt" executable in your $PATH.
If not specified it defaults to "rustfmt":

```
let g:rustfmt_command = 'rustfmt'
```

g:rustfmt_autosave

b:rustfmt_autosave

g:rustfmt_autosave
Set this option to 1 to run :RustFmt automatically when saving a
buffer. If not specified it defaults to 0:

```
let g:rustfmt_autosave = 0
```

There is also a buffer-local b:rustfmt_autosave that can be set for
the same purpose, and can override the global setting.

g:rustfmt_autosave_if_config_present

b:rustfmt_autosave_if_config_present

g:rustfmt_autosave_if_config_present
Set this option to 1 to have b:rustfmt_autosave be set automatically
if a rustfmt.toml file is present in any parent directory leading to
the file being edited. If not set, default to 0:

```
let g:rustfmt_autosave_if_config_present = 0
```

This is useful to have rustfmt only execute on save, on projects
that have rustfmt.toml configuration.

There is also a buffer-local b:rustfmt_autosave_if_config_present
that can be set for the same purpose, which can override the global
setting.

g:rustfmt_fail_silently

g:rustfmt_fail_silently
Set this option to 1 to prevent "rustfmt" from populating the
location-list with errors. If not specified it defaults to 0:

```
let g:rustfmt_fail_silently = 0
```

g:rustfmt_options

g:rustfmt_options
Set this option to a string of options to pass to "rustfmt".
The write-mode is already set to "overwrite". If not specified it
defaults to '':

```
let g:rustfmt_options = ''
```

g:rustfmt_emit_files

g:rustfmt_emit_files
	If not specified rust.vim tries to detect the right parameter to
	pass to rustfmt based on its reported version. Otherwise, it
	determines whether to run rustfmt with "--emit=files" (when 1 is
	provided) instead of "--write-mode=overwrite".

```
let g:rustfmt_emit_files = 0
```

g:rustfmt_detect_version

g:rustfmt_detect_version
	When set to 1, will try to parse the version output from "rustfmt".
	Disabled by default for performance reasons.

```
let g:rustfmt_detect_version = 1
```

g:rustfmt_find_toml

g:rustfmt_find_toml
	When set to 1, will try to find rustfmt.toml file by searching from
	current path upwards. Disabled by default for performance reasons.

```
let g:rustfmt_find_toml = 1
```

g:rust_playpen_url

g:rust_playpen_url
	Set this option to override the url for the playpen to use:

```
let g:rust_playpen_url = 'https://play.rust-lang.org/'
```

g:rust_shortener_url

g:rust_shortener_url
	Set this option to override the url for the url shortener:

```
let g:rust_shortener_url = 'https://is.gd/'
```

g:rust_clip_command

g:rust_clip_command
	Set this option to the command used in your OS to copy the Rust Play
	url to the clipboard:

```
let g:rust_clip_command = 'xclip -selection clipboard'
```

g:cargo_makeprg_params

g:cargo_makeprg_params
	Set this option to the string of parameters to pass to cargo. If not
	specified it defaults to $*:

```
let g:cargo_makeprg_params = 'build'
```

g:cargo_shell_command_runner

g:cargo_shell_command_runner
	Set this option to change how to run shell commands for cargo commands
	:Cargo, :Cbuild, :Crun, etc. - see rust-commands.
	By default, :terminal is used to run a shell command in a terminal
	window asynchronously. But if you prefer :! for running the
	commands, it can be specified:

```
let g:cargo_shell_command_runner = '!'
```

Integration with Syntastic

rust-syntastic

This plugin automatically integrates with the Syntastic checker. There are
two checkers provided: rustc, and cargo. The latter invokes cargo to
build code, and the former delivers a single edited ".rs" file as a
compilation target directly to the Rust compiler, rustc.

Because Cargo is almost exclusively being used for building Rust code these
days, cargo is the default checker.

```
let g:syntastic_rust_checkers = ['cargo']
```

If you would like to change it, you can set g:syntastic_rust_checkers to a
different value.

g:rust_cargo_avoid_whole_workspace

b:rust_cargo_avoid_whole_workspace

g:rust_cargo_avoid_whole_workspace
	When editing a crate that is part of a Cargo workspace, and this
	option is set to 1 (the default), then cargo will be executed
	directly in that crate directory instead of in the workspace
	directory. Setting 0 prevents this behavior - however be aware that
	if you are working in a large workspace, Cargo commands may take more

time, plus the Syntastic error list may include all the crates in the workspace.

```
let g:rust_cargo_avoid_whole_workspace = 0
```

g:rust_cargo_check_all_targets

b:rust_cargo_check_all_targets

g:rust_cargo_check_all_targets
When set to 1, the --all-targets option will be passed to cargo when Syntastic executes it, allowing the linting of all targets under the package.
The default is 0.

g:rust_cargo_check_all_features

b:rust_cargo_check_all_features

g:rust_cargo_check_all_features
When set to 1, the --all-features option will be passed to cargo when Syntastic executes it, allowing the linting of all features of the package.
The default is 0.

g:rust_cargo_check_examples

b:rust_cargo_check_examples

g:rust_cargo_check_examples
When set to 1, the --examples option will be passed to cargo when Syntastic executes it, to prevent the exclusion of examples from linting. The examples are normally under the examples/ directory of the crate.
The default is 0.

g:rust_cargo_check_tests

b:rust_cargo_check_tests

g:rust_cargo_check_tests
When set to 1, the --tests option will be passed to cargo when Syntastic executes it, to prevent the exclusion of tests from linting. The tests are normally under the tests/ directory of the crate.
The default is 0.

g:rust_cargo_check_benches

b:rust_cargo_check_benches

g:rust_cargo_check_benches
When set to 1, the --benches option will be passed to cargo when Syntastic executes it. The benches are normally under the benches/ directory of the crate.
The default is 0.

Integration with auto-pairs

rust-auto-pairs

This plugin automatically configures the auto-pairs plugin not to duplicate single quotes, which are used more often for lifetime annotations than for single character literals.

g:rust_keep_autopairs_default

g:rust_keep_autopairs_default
Don't override auto-pairs default for the Rust filetype. The default is 0.

COMMANDS

rust-commands

Invoking Cargo

This plugin defines very simple shortcuts for invoking Cargo from within Vim.

:Cargo <args>

:Cargo

Runs cargo with the provided arguments.

:Cbuild <args>

:Cbuild

Shortcut for cargo build.

:Cclean <args>

:Cclean

Shortcut for cargo clean.

:Cdoc <args>

:Cdoc

Shortcut for cargo doc.

:Cinit <args>

:Cinit

Shortcut for cargo init.

:Crun <args>

:Crun

Shortcut for cargo run.

:Ctest <args> :Ctest

Shortcut for cargo test.

:Cupdate <args> :Cupdate

Shortcut for cargo update.

:Cbench <args> :Cbench

Shortcut for cargo bench.

:Csearch <args> :Csearch

Shortcut for cargo search.

:Cpublish <args> :Cpublish

Shortcut for cargo publish.

:Cinstall <args> :Cinstall

Shortcut for cargo install.

:Cruntarget <args> :Cruntarget

Shortcut for cargo run --bin or cargo run --example,
depending on the currently open buffer.

Formatting

:RustFmt :RustFmt

Runs g:rustfmt_command on the current buffer. If
g:rustfmt_options is set then those will be passed to the
executable.

If g:rustfmt_fail_silently is 0 (the default) then it
will populate the location-list with the errors from
g:rustfmt_command. If g:rustfmt_fail_silently is set to 1
then it will not populate the location-list.

:RustFmtRange :RustFmtRange

Runs g:rustfmt_command with selected range. See :RustFmt
for any other information.

Playpen integration

:RustPlay :RustPlay

This command will only work if you have web-api.vim installed
(available at https://github.com/mattn/webapi-vim). It sends
the current selection, or if nothing is selected, the entirety
of the current buffer to the Rust playpen, and emits a message
with the shortened URL to the playpen.

g:rust_playpen_url is the base URL to the playpen, by default
"https://play.rust-lang.org/".

g:rust_shortener_url is the base url for the shortener, by
default "https://is.gd/"

g:rust_clip_command is the command to run to copy the
playpen url to the clipboard of your system.

Evaluation of a single Rust file

NOTE: These commands are useful only when working with standalone Rust files,
which is usually not the case for common Rust development. If you wish to
build Rust crates from within Vim, use Vim's make, Syntastic, or functionality
from other plugins.

:RustRun [args] :RustRun

:RustRun! [rustc-args] [--] [args]
Compiles and runs the current file. If it has unsaved
changes, it will be saved first using :update. If the
current file is an unnamed buffer, it will be written to a
temporary file first. The compiled binary is always placed in
a temporary directory, but is run from the current directory.

The arguments given to :RustRun will be passed to the
compiled binary.

If ! is specified, the arguments are passed to rustc instead.
A "--" argument will separate the rustc arguments from the
arguments passed to the binary.

If g:rustc_path is defined, it is used as the path to rustc.
Otherwise it is assumed rustc can be found in $PATH.

:RustExpand [args] :RustExpand

:RustExpand! [TYPE] [args]
Expands the current file using --pretty and displays the
results in a new split. If the current file has unsaved
changes, it will be saved first using :update. If the
current file is an unnamed buffer, it will be written to a
temporary file first.

The arguments given to :RustExpand will be passed to rustc.
This is largely intended for specifying various --cfg

configurations.

If ! is specified, the first argument is the expansion type to pass to rustc --pretty. Otherwise it will default to "expanded".

If g:rustc_path is defined, it is used as the path to rustc. Otherwise it is assumed rustc can be found in $PATH.

:RustEmitIr [args] :RustEmitIr

Compiles the current file to LLVM IR and displays the results in a new split. If the current file has unsaved changes, it will be saved first using :update. If the current file is an unnamed buffer, it will be written to a temporary file first.

The arguments given to :RustEmitIr will be passed to rustc.

If g:rustc_path is defined, it is used as the path to rustc. Otherwise it is assumed rustc can be found in $PATH.

:RustEmitAsm [args] :RustEmitAsm

Compiles the current file to assembly and displays the results in a new split. If the current file has unsaved changes, it will be saved first using :update. If the current file is an unnamed buffer, it will be written to a temporary file first.

The arguments given to :RustEmitAsm will be passed to rustc.

If g:rustc_path is defined, it is used as the path to rustc. Otherwise it is assumed rustc can be found in $PATH.

Running test(s)

:[N]RustTest[!] [options] :RustTest

Runs a test under the cursor when the current buffer is in a cargo project with "cargo test" command. If the command did not find any test function under the cursor, it stops with an error message.

When N is given, adjust the size of the new window to N lines or columns.

When ! is given, runs all tests regardless of current cursor position.

When [options] is given, it is passed to "cargo" command arguments.

When the current buffer is outside a cargo project, the command runs "rustc --test" command instead of "cargo test" as fallback. All tests are run regardless of adding ! since there is no way to run specific test function with rustc. [options] is passed to "rustc" command arguments in that case.

Takes optional modifiers (see <mods>):

```
:tab RustTest
:belowright 16RustTest
:leftabove vert 80RustTest
```

rust.vim Debugging

:RustInfo :RustInfo

Emits debugging info of the Vim Rust plugin.

:RustInfoToClipboard :RustInfoClipboard

Saves debugging info of the Vim Rust plugin to the default register.

:RustInfoToFile [filename] :RustInfoToFile

Saves debugging info of the Vim Rust plugin to the given file, overwriting it.

MAPPINGS rust-mappings

This plugin defines Normal mode, Visual mode, and Operator-pending mode mappings for [[and]] to support hanging indents.

by David Fishburn

This is a filetype plugin to work with SQL files.

The Structured Query Language (SQL) is a standard which specifies statements that allow a user to interact with a relational database. Vim includes features for navigation, indentation and syntax highlighting.

1. Navigation

The SQL ftplugin provides a number of options to assist with file navigation.

1.1 Matchit

sql-matchit

The matchit plugin (https://www.vim.org/scripts/script.php?script_id=39) provides many additional features and can be customized for different languages. The matchit plugin is configured by defining a local buffer variable, b:match_words. Pressing the % key while on various keywords will move the cursor to its match. For example, if the cursor is on an "if", pressing % will cycle between the "else", "elseif" and "end if" keywords.

The following keywords are supported:

```
if
elseif | elsif
else [if]
end if

[while condition] loop
    leave
    break
    continue
    exit
end loop

for
    leave
    break
    continue
    exit
end loop

do
    statements
doend

case
when
when
default
end case

merge
when not matched
when matched

create[ or replace] procedure|function|event
returns
```

1.2 Text Object Motions

sql-object-motions

Vim has a number of predefined keys for working with text object-motions. This filetype plugin attempts to translate these keys to maps which make sense for the SQL language.

The following Normal mode and Visual mode maps exist (when you edit a SQL file):

```
]]                  move forward to the next 'begin'
[[                  move backwards to the previous 'begin'
][                  move forward to the next 'end'
[]                  move backwards to the previous 'end'
```

1.3 Predefined Object Motions

sql-predefined-objects

Most relational databases support various standard features, tables, indices, triggers and stored procedures. Each vendor also has a variety of proprietary objects. The next set of maps have been created to help move between these objects. Depends on which database vendor you are using, the list of objects must be configurable. The filetype plugin attempts to define many of the standard objects, plus many additional ones. In order to make this as flexible as possible, you can override the list of objects from within your vimrc with the following:

```
let g:ftplugin_sql_objects = 'function,procedure,event,table,trigger' ..
            \ ',schema,service,publication,database,datatype,domain' ..
            \ ',index,subscription,synchronization,view,variable'
```

The following Normal mode and Visual mode maps have been created which use
the above list:

```
]}                  move forward to the next 'create <object name>'
[{                  move backward to the previous 'create <object name>'
```

Repeatedly pressing]} will cycle through each of these create statements:

```
create table t1 (
    ...
);

create procedure p1
begin
    ...
end;

create index i1 on t1 (c1);
```

The default setting for g:ftplugin_sql_objects is:

```
let g:ftplugin_sql_objects = 'function,procedure,event,' ..
            \ '\\(existing\\\\|global\\s\\+temporary\\s\\+\\)\\\{,1}' ..
            \ 'table,trigger' ..
            \ ',schema,service,publication,database,datatype,domain' ..
            \ ',index,subscription,synchronization,view,variable'
```

The above will also handle these cases:

```
create table t1 (
    ...
);
create existing table t2 (
    ...
);
create global temporary table t3 (
    ...
);
```

By default, the ftplugin only searches for CREATE statements. You can also
override this via your init.vim with the following:

```
let g:ftplugin_sql_statements = 'create,alter'
```

The filetype plugin defines three types of comments:

```
1.  --
2.  //
3.  /*
     *
     */
```

The following Normal mode and Visual mode maps have been created to work
with comments:

```
]"                  move forward to the beginning of a comment
["                  move forward to the end of a comment
```

1.4 Macros

sql-macros

Vim's feature to find macro definitions, 'define', is supported using this
regular expression:

```
\c\<\(VARIABLE\|DECLARE\|IN\|OUT\|INOUT\)\>
```

This addresses the following code:

```
CREATE VARIABLE myVar1 INTEGER;

CREATE PROCEDURE sp_test(
    IN myVar2 INTEGER,
    OUT myVar3 CHAR(30),
    INOUT myVar4 NUMERIC(20,0)
)
BEGIN
    DECLARE myVar5 INTEGER;

    SELECT c1, c2, c3
      INTO myVar2, myVar3, myVar4
      FROM T1
     WHERE c4 = myVar1;
END;
```

Place your cursor on "myVar1" on this line:

```
WHERE c4 = myVar1;
           ^
```

Press any of the following keys:

```
[d
[D
[CTRL-D
```

2. SQL Dialects

sql-dialects sql-types

sybase TSQL Transact-SQL

sqlanywhere

oracle plsql sqlj

sqlserver

mysql postgresql psql

informix

All relational databases support SQL. There is a portion of SQL that is portable across vendors (ex. CREATE TABLE, CREATE INDEX), but there is a great deal of vendor specific extensions to SQL. Oracle supports the "CREATE OR REPLACE" syntax, column defaults specified in the CREATE TABLE statement and the procedural language (for stored procedures and triggers).

The default Vim distribution ships with syntax highlighting based on Oracle's PL/SQL. The default SQL indent script works for Oracle and SQL Anywhere. The default filetype plugin works for all vendors and should remain vendor neutral, but extendable.

Vim currently has support for a variety of different vendors, currently this is via syntax scripts. Unfortunately, to flip between different syntax rules you must either create:

1. New filetypes
2. Custom autocmds
3. Manual steps / commands

The majority of people work with only one vendor's database product, it would be nice to specify a default in your init.vim.

2.1 SQLSetType

sqlsettype SQLSetType

For the people that work with many different databases, it is nice to be able to flip between the various vendors rules (indent, syntax) on a per buffer basis, at any time. The ftplugin/sql.vim file defines this function:

```
SQLSetType
```

Executing this function without any parameters will set the indent and syntax scripts back to their defaults, see sql-type-default. You can use the <Tab> key to complete the optional parameter.

After typing the function name and a space, you can use the completion to supply a parameter. The function takes the name of the Vim script you want to source. Using the cmdline-completion feature, the SQLSetType function will search the 'runtimepath' for all Vim scripts with a name containing "sql". This takes the guess work out of the spelling of the names. The following are examples:

```
:SQLSetType
:SQLSetType sqloracle
:SQLSetType sqlanywhere
:SQLSetType sqlinformix
:SQLSetType mysql
```

The easiest approach is to the use <Tab> character which will first complete the command name (SQLSetType), after a space and another <Tab>, display a list of available Vim script names:

```
:SQL<Tab><space><Tab>
```

2.2 SQLGetType

sqlgettype SQLGetType

At anytime you can determine which SQL dialect you are using by calling the SQLGetType command. The ftplugin/sql.vim file defines this function:

```
SQLGetType
```

This will echo:

```
Current SQL dialect in use:sqlanywhere
```

2.3 SQL Dialect Default

sql-type-default

As mentioned earlier, the default syntax rules for Vim is based on Oracle (PL/SQL). You can override this default by placing one of the following in your init.vim:

```
let g:sql_type_default = 'sqlanywhere'
let g:sql_type_default = 'sqlinformix'
let g:sql_type_default = 'mysql'
```

If you added the following to your init.vim:

```
let g:sql_type_default = 'sqlinformix'
```

The next time edit a SQL file the following scripts will be automatically loaded by Vim:

```
ftplugin/sql.vim
syntax/sqlinformix.vim
indent/sql.vim
```

Notice indent/sqlinformix.sql was not loaded. There is no indent file for Informix, Vim loads the default files if the specified files does not exist.

3. Adding new SQL Dialects

sql-adding-dialects

If you begin working with a SQL dialect which does not have any customizations available with the default Vim distribution you can check https://www.vim.org to see if any customization currently exist. If not, you can begin by cloning an existing script. Read filetype-plugins for more details.

To help identify these scripts, try to create the files with a "sql" prefix. If you decide you wish to create customizations for the SQLite database, you can create any of the following:

```
Unix
    ~/.config/nvim/syntax/sqlite.vim
    ~/.config/nvim/indent/sqlite.vim
```

No changes are necessary to the SQLSetType function. It will automatically pick up the new SQL files and load them when you issue the SQLSetType command.

4. OMNI SQL Completion

sql-completion

omni-sql-completion

Vim 7 includes a code completion interface and functions which allows plugin developers to build in code completion for any language. Vim 7 includes code completion for the SQL language.

There are two modes to the SQL completion plugin, static and dynamic. The static mode populates the popups with the data generated from current syntax highlight rules. The dynamic mode populates the popups with data retrieved directly from a database. This includes, table lists, column lists, procedures names and more.

4.1 Static Mode

sql-completion-static

The static popups created contain items defined by the active syntax rules
while editing a file with a filetype of SQL. The plugin defines (by default)
various maps to help the user refine the list of items to be displayed.
The defaults static maps are:

```
imap <buffer> <C-C>a <C-\><C-O>:call sqlcomplete#Map('syntax')<CR><C-X><C-O>
imap <buffer> <C-C>k <C-\><C-O>:call sqlcomplete#Map('sqlKeyword')<CR><C-X><C-O>
imap <buffer> <C-C>f <C-\><C-O>:call sqlcomplete#Map('sqlFunction')<CR><C-X><C-O>
imap <buffer> <C-C>o <C-\><C-O>:call sqlcomplete#Map('sqlOption')<CR><C-X><C-O>
imap <buffer> <C-C>T <C-\><C-O>:call sqlcomplete#Map('sqlType')<CR><C-X><C-O>
imap <buffer> <C-C>s <C-\><C-O>:call sqlcomplete#Map('sqlStatement')<CR><C-X><C-O>
```

The use of "<C-C>" can be user chosen by using the following in your init.vim
as it may not work properly on all platforms:

```
let g:ftplugin_sql_omni_key = '<C-C>'
```

The static maps (which are based on the syntax highlight groups) follow this
format:

```
imap <buffer> <C-C>k <C-\><C-O>:call sqlcomplete#Map('sqlKeyword')<CR><C-X><C-O>
imap <buffer> <C-C>k <C-\><C-O>:call sqlcomplete#Map('sqlKeyword\w*')<CR><C-X><C-O>
```

This command breaks down as:

```
imap               - Create an insert map
<buffer>                   - Only for this buffer
<C-C>k             - Your choice of key map
<C-\><C-O>                 - Execute one command, return to Insert mode
:call sqlcomplete#Map( - Allows the SQL completion plugin to perform some
                         housekeeping functions to allow it to be used in
                         conjunction with other completion plugins.
                         Indicate which item you want the SQL completion
                         plugin to complete.
                         In this case we are asking the plugin to display
                         items from the syntax highlight group
                         'sqlKeyword'.
                         You can view a list of highlight group names to
                         choose from by executing the
                             :syntax list
                         command while editing a SQL file.
'sqlKeyword'       - Display the items for the sqlKeyword highlight
                         group
'sqlKeyword\w*'    - A second option available with Vim 7.4 which
                         uses a regular expression to determine which
                         syntax groups to use
)<CR>              - Execute the :let command
<C-X><C-O>                 - Trigger the standard omni completion key stroke.
                         Passing in 'sqlKeyword' instructs the SQL
                         completion plugin to populate the popup with
                         items from the sqlKeyword highlight group.  The
                         plugin will also cache this result until Vim is
                         restarted.  The syntax list is retrieved using
                         the syntaxcomplete plugin.
```

Using the 'syntax' keyword is a special case. This instructs the
syntaxcomplete plugin to retrieve all syntax items. So this will effectively
work for any of Vim's SQL syntax files. At the time of writing this includes
10 different syntax files for the different dialects of SQL (see section 3
above, sql-dialects).

Here are some examples of the entries which are pulled from the syntax files:

```
All
    - Contains the contents of all syntax highlight groups
Statements
    - Select, Insert, Update, Delete, Create, Alter, ...
Functions
    - Min, Max, Trim, Round, Date, ...
Keywords
    - Index, Database, Having, Group, With
Options
    - Isolation_level, On_error, Qualify_owners, Fire_triggers, ...
Types
    - Integer, Char, Varchar, Date, DateTime, Timestamp, ...
```

4.2 Dynamic Mode

sql-completion-dynamic

Dynamic mode populates the popups with data directly from a database. In
order for the dynamic feature to be enabled you must have the dbext.vim
plugin installed, (https://www.vim.org/scripts/script.php?script_id=356).

Dynamic mode is used by several features of the SQL completion plugin.
After installing the dbext plugin see the dbext-tutorial for additional

configuration and usage. The dbext plugin allows the SQL completion plugin
to display a list of tables, procedures, views and columns.

```
Table List
    - All tables for all schema owners
Procedure List
    - All stored procedures for all schema owners
View List
    - All stored procedures for all schema owners
Column List
    - For the selected table, the columns that are part of the table
```

To enable the popup, while in INSERT mode, use the following key combinations
for each group (where <C-C> means hold the CTRL key down while pressing
the space bar):

```
    Table List            - <C-C>t
```

- <C-X><C-O> (the default map assumes tables)

```
       Stored Procedure List - <C-C>p
       View List             - <C-C>v
       Column List           - <C-C>c
    Drilling In / Out     - When viewing a popup window displaying the list
                            of tables, you can press <Right>, this will
                            replace the table currently highlighted with
                            the column list for that table.
```

- When viewing a popup window displaying the list

```
                              of columns, you can press <Left>, this will
                              replace the column list with the list of tables.
```

- This allows you to quickly drill down into a

```
                              table to view its columns and back again.
```

- <Right> and <Left> can also be chosen via

```
                              your init.vim
```

```
let g:ftplugin_sql_omni_key_right = '<Right>'
let g:ftplugin_sql_omni_key_left  = '<Left>'
```

The SQL completion plugin caches various lists that are displayed in
the popup window. This makes the re-displaying of these lists very
fast. If new tables or columns are added to the database it may become
necessary to clear the plugins cache. The default map for this is:

```
imap <buffer> <C-C>R <C-\><C-O>:call sqlcomplete#Map('ResetCache')<CR><C-X><C-O>
```

4.3 SQL Tutorial sql-completion-tutorial

This tutorial is designed to take you through the common features of the SQL
completion plugin so that:

```
a) You gain familiarity with the plugin
b) You are introduced to some of the more common features
c) Show how to customize it to your preferences
d) Demonstrate "Best of Use" of the plugin (easiest way to configure).
```

First, create a new buffer:

```
:e tutorial.sql
```

Static features

To take you through the various lists, simply enter insert mode, hit:

```
    <C-C>s   (show SQL statements)
```

At this point, you can page down through the list until you find "select".
If you are familiar with the item you are looking for, for example you know
the statement begins with the letter "s". You can type ahead (without the
quotes) "se" then press:

```
    <C-Space>t
```

Assuming "select" is highlighted in the popup list press <Enter> to choose
the entry. Now type:

```
    * fr<C-C>a (show all syntax items)
```

choose "from" from the popup list.

When writing stored procedures using the "type" list is useful. It contains
a list of all the database supported types. This may or may not be true
depending on the syntax file you are using. The SQL Anywhere syntax file
(sqlanywhere.vim) has support for this:

```
BEGIN
   DECLARE customer_id <C-C>T <-- Choose a type from the list
```

Dynamic features

To take advantage of the dynamic features you must first install the dbext.vim plugin (https://www.vim.org/scripts/script.php?script_id=356). It also comes with a tutorial. From the SQL completion plugin's perspective, the main feature dbext provides is a connection to a database. dbext connection profiles are the most efficient mechanism to define connection information. Once connections have been setup, the SQL completion plugin uses the features of dbext in the background to populate the popups.

What follows assumes dbext.vim has been correctly configured, a simple test is to run the command, :DBListTable. If a list of tables is shown, you know dbext.vim is working as expected. If not, please consult the dbext.txt documentation.

Assuming you have followed the dbext-tutorial you can press <C-C>t to display a list of tables. There is a delay while dbext is creating the table list. After the list is displayed press <C-W>. This will remove both the popup window and the table name already chosen when the list became active.

4.3.1 Table Completion: sql-completion-tables

Press <C-C>t to display a list of tables from within the database you have connected via the dbext plugin.
NOTE: All of the SQL completion popups support typing a prefix before pressing the key map. This will limit the contents of the popup window to just items beginning with those characters.

4.3.2 Column Completion: sql-completion-columns

The SQL completion plugin can also display a list of columns for particular tables. The column completion is triggered via <C-C>c.

NOTE: The following example uses <Right> to trigger a column list while the popup window is active.

Example of using column completion:

- Press <C-C>t again to display the list of tables.
- When the list is displayed in the completion window, press <Right>, this will replace the list of tables, with a list of columns for the table highlighted (after the same short delay).
- If you press <Left>, this will again replace the column list with the list of tables. This allows you to drill into tables and column lists very quickly.
- Press <Right> again while the same table is highlighted. You will notice there is no delay since the column list has been cached. If you change the schema of a cached table you can press <C-C>R, which clears the SQL completion cache.
- **NOTE:** <Right> and <Left> have been designed to work while the completion window is active. If the completion popup window is not active, a normal <Right> or <Left> will be executed.

Let's look at how we can build a SQL statement dynamically. A select statement requires a list of columns. There are two ways to build a column list using the SQL completion plugin.

```
One column at a time:
```

1. After typing SELECT press <C-C>t to display a list of tables.
2. Choose a table from the list.
3. Press <Right> to display a list of columns.
4. Choose the column from the list and press enter.
5. Enter a "," and press <C-C>c. Generating a column list generally requires having the cursor on a table name. The plugin uses this name to determine what table to retrieve the column list. In this step, since we are pressing <C-C>c without the cursor on a table name the column list displayed will be for the previous table. Choose a different column and move on.
6. Repeat step 5 as often as necessary.

```
All columns for a table:
```

7. After typing SELECT press <C-C>t to display a list of tables.

8. Highlight the table you need the column list for.

9. Press <Enter> to choose the table from the list.

10. Press <C-C>l to request a comma-separated list of all columns
    for this table.

11. Based on the table name chosen in step 3, the plugin attempts to
    decide on a reasonable table alias. You are then prompted to
    either accept of change the alias. Press OK.

12. The table name is replaced with the column list of the table is
    replaced with the comma separate list of columns with the alias
    prepended to each of the columns.

13. Step 3 and 4 can be replaced by pressing <C-C>L, which has
    a <C-Y> embedded in the map to choose the currently highlighted
    table in the list.

There is a special provision when writing select statements. Consider the
following statement:

```
select *
  from customer c,
       contact cn,
       department as dp,
       employee e,
       site_options so
 where c.
```

In INSERT mode after typing the final "c." which is an alias for the
"customer" table, you can press either <C-C>c or <C-X><C-O>. This will
popup a list of columns for the customer table. It does this by looking back
to the beginning of the select statement and finding a list of the tables
specified in the FROM clause. In this case it notes that in the string
"customer c", "c" is an alias for the customer table. The optional "AS"
keyword is also supported, "customer AS c".

4.3.3 Procedure Completion: sql-completion-procedures

Similar to the table list, <C-C>p, will display a list of stored
procedures stored within the database.

4.3.4 View Completion: sql-completion-views

Similar to the table list, <C-C>v, will display a list of views in the
database.

4.4 Completion Customization sql-completion-customization

The SQL completion plugin can be customized through various options set in
your init.vim:

```
omni_sql_no_default_maps
```

- Default: This variable is not defined

- If this variable is defined, no maps are created for OMNI
  completion. See sql-completion-maps for further discussion.

```
omni_sql_use_tbl_alias
```

- Default: a

- This setting is only used when generating a comma-separated
  column list. By default the map is <C-C>l. When generating
  a column list, an alias can be prepended to the beginning of each
  column, for example: e.emp_id, e.emp_name. This option has three
  settings:

```
n - do not use an alias
d - use the default (calculated) alias
a - ask to confirm the alias name
```

An alias is determined following a few rules:

1. If the table name has an '_', then use it as a separator:

```
MY_TABLE_NAME --> MTN
my_table_name --> mtn
My_table_NAME --> MtN
```

2. If the table name does NOT contain an '_', but DOES use mixed case then the case is used as a separator:

```
MyTableName --> MTN
```

3. If the table name does NOT contain an '_', and does NOT use mixed case then the first letter of the table is used:

```
mytablename --> m
MYTABLENAME --> M

omni_sql_ignorecase
```

- Default: Current setting for 'ignorecase'
- Valid settings are 0 or 1.
- When entering a few letters before initiating completion, the list will be filtered to display only the entries which begin with the list of characters. When this option is set to 0, the list will be filtered using case sensitivity.

```
omni_sql_include_owner
```

- Default: 0, unless dbext.vim 3.00 has been installed
- Valid settings are 0 or 1.
- When completing tables, procedure or views and using dbext.vim 3.00 or higher the list of objects will also include the owner name. When completing these objects and omni_sql_include_owner is enabled the owner name will be replaced.

```
omni_sql_precache_syntax_groups
```

- Default:

```
['syntax','sqlKeyword','sqlFunction','sqlOption','sqlType','sqlStatement']
```

- sqlcomplete can be used in conjunction with other completion plugins. This is outlined at sql-completion-filetypes. When the filetype is changed temporarily to SQL, the sqlcompletion plugin will cache the syntax groups listed in the List specified in this option.

4.5 SQL Maps

sql-completion-maps

The default SQL maps have been described in other sections of this document in greater detail. Here is a list of the maps with a brief description of each.

Static Maps

These are maps which use populate the completion list using Vim's syntax highlighting rules.

```
<C-C>a
```

- Displays all SQL syntax items.

```
<C-C>k
```

- Displays all SQL syntax items defined as 'sqlKeyword'.

  ```
  <C-C>f
  ```

- Displays all SQL syntax items defined as 'sqlFunction.

  ```
  <C-C>o
  ```

- Displays all SQL syntax items defined as 'sqlOption'.

  ```
  <C-C>T
  ```

- Displays all SQL syntax items defined as 'sqlType'.

  ```
  <C-C>s
  ```

- Displays all SQL syntax items defined as 'sqlStatement'.

  ```
  ```

Dynamic Maps ~

These are maps which use populate the completion list using the dbext.vim plugin.

```
<C-C>t
```

- Displays a list of tables.

  ```
  <C-C>p
  ```

  - Displays a list of procedures.

    ```
    <C-C>v
    ```

  - Displays a list of views.

    ```
    <C-C>c
    ```

  - Displays a list of columns for a specific table.

    ```
    <C-C>l
    ```

  - Displays a comma-separated list of columns for a specific table.

    ```
    <C-C>L
    ```

  - Displays a comma-separated list of columns for a specific table.

    This should only be used when the completion window is active.

    ```
    <Right>
    ```

- Displays a list of columns for the table currently highlighted in

  the completion window. <Right> is not recognized on most Unix systems, so this maps is only created on the Windows platform. If you would like the same feature on Unix, choose a different key and make the same map in your vimrc.

  ```
  <Left>
  ```

- Displays the list of tables.

  <Left> is not recognized on most Unix systems, so this maps is only created on the Windows platform. If you would like the same feature on Unix, choose a different key and make the same map in your vimrc.

  ```
  <C-C>R
  ```

- This maps removes all cached items and forces the SQL completion

  to regenerate the list of items.

Customizing Maps

You can create as many additional key maps as you like. Generally, the maps will be specifying different syntax highlight groups.

If you do not wish the default maps created or the key choices do not work on your platform (often a case on unix) you define the following variable in your init.vim:

```
let g:omni_sql_no_default_maps = 1
```

Do not edit ftplugin/sql.vim directly! If you change this file your changes will be over written on future updates. Vim has a special directory structure which allows you to make customizations without changing the files that are included with the Vim distribution. If you wish to customize the maps create an after/ftplugin/sql.vim (see after-directory) and place the same maps from the ftplugin/sql.vim in it using your own key strokes. <C-C> was chosen since it will work on both Windows and unix platforms. On the windows platform you can also use <C-Space> or ALT keys.

4.6 Using with other filetypes

sql-completion-filetypes

Many times SQL can be used with different filetypes. For example Perl, Java, PHP, Javascript can all interact with a database. Often you need both the SQL completion and the completion capabilities for the current language you are editing.

This can be enabled easily with the following steps (assuming a Perl file):

```
1.  :e test.pl
2.  :set filetype=sql
3.  :set ft=perl
```

Step 1

Begins by editing a Perl file. Vim automatically sets the filetype to "perl". By default, Vim runs the appropriate filetype file ftplugin/perl.vim. If you are using the syntax completion plugin by following the directions at ft-syntax-omni then the 'omnifunc' option has been set to "syntax#Complete". Pressing <C-X><C-O> will display the omni popup containing the syntax items for Perl.

Step 2

Manually setting the filetype to "sql" will also fire the appropriate filetype files ftplugin/sql.vim. This file will define a number of buffer specific maps for SQL completion, see sql-completion-maps. Now these maps have been created and the SQL completion plugin has been initialized. All SQL syntax items have been cached in preparation. The SQL filetype script detects we are attempting to use two different completion plugins. Since the SQL maps begin with <C-C>, the maps will toggle the 'omnifunc' when in use. So you can use <C-X><C-O> to continue using the completion for Perl (using the syntax completion plugin) and <C-C> to use the SQL completion features.

Step 3

Setting the filetype back to Perl sets all the usual "perl" related items back as they were.

Nvim Graphical User Interface GUI

Any client that supports the Nvim ui-protocol can be used as a UI for Nvim. And multiple UIs can connect to the same Nvim instance! The terms "UI" and "GUI" are often used interchangeably because all Nvim UI clients have the same potential capabilities; the "TUI" refers to a UI client that outputs to your terminal, whereas a "GUI" outputs directly to the OS graphics system.

Except where noted, this document describes UI capabilities available to both TUI and GUI (assuming the UI supports the given feature). See TUI for notes specific to the terminal UI. Help tags with the "gui-" prefix refer to UI features, whereas help tags with the "ui-" prefix refer to the ui-protocol.

Third-party GUIs

third-party-guis vscode

Nvim provides a builtin "terminal UI" (TUI), but also works with many (third-party) GUIs which may provide a fresh look or extra features on top of Nvim. For example, "vscode-neovim" essentially allows you to use VSCode as a Nvim GUI.

- vscode-neovim (Nvim in VSCode!) https://github.com/vscode-neovim/vscode-neovim
- Firenvim (Nvim in your web browser!) https://github.com/glacambre/firenvim
- Neovide https://neovide.dev/
- Goneovim https://github.com/akiyosi/goneovim
- Nvy https://github.com/RMichelsen/Nvy
- Neovim-Qt (Qt5) https://github.com/equalsraf/neovim-qt
- VimR (macOS) https://github.com/qvacua/vimr
- Others https://github.com/neovim/neovim/wiki/Related-projects#gui

Starting the GUI

gui-config gui-start

ginit.vim gui-init gvimrc $MYGVIMRC For GUI-specific configuration Nvim provides the UIEnter event. This happens after other initializations, or whenever a UI attaches (multiple UIs can connect to any Nvim instance).

Example: this sets "g:gui" to the value of the UI's "rgb" field:

```
:autocmd UIEnter * let g:gui = filter(nvim_list_uis(),{k,v-> v.chan==v:event.chan})[0].rgb
```

ui-lifecycle

Nvim's clientserver architecture allows you to :restart, :detach (Nvim continues in the background), or :connect to another instance.

Use cases:

- keep the session running on remote machines after (unexpected) SSH disconnect
- open files in an existing session from your OS file explorer
- run Nvim as a background process (daemon/service), atttach/detach as needed

Stop or detach the current UI

:detach E5768 :[range]detach Detaches the current UI. Other UIs (if any) remain attached, unless [range] was given. If [range] is "%", all UIs _except_ the current UI are detached:

```
:%detach
```

The server (typically `nvim --embed`) continues running as a background process. Use :connect or --remote-ui to reattach. Before detaching, you may want to note the server address:

```
:echo v:servername
```

Note: The server closes the UI RPC channel, so :detach works for any GUI/UI; but if the client isn't expecting this, it may (incorrectly) report an error.

:detach!

:detach! Sets the current UI as "detachable" without detaching it now: the UI stays attached, but if it later disconnects unexpectedly (e.g. host terminal closed, ssh disconnect, etc.), the server keeps running in the background. Use :connect or --remote-ui to reattach.

Same as nvim_chan_set() `detach=true` on the UI channel. Preserved across :restart.

If you always want Nvim to survive accidental disconnect, put this in your config:

```
silent detach!
```

Restart Nvim

:restart :restart! E5201 :restart[!] [+cmd] [command] Restarts Nvim. Saves and restores the current session, unless [!] is given. Sets v:exitreason. See also ZR.

1. Saves the current session (unless [!] was given).
2. Stops Nvim using `:qall` (or +cmd, if given).
3. Starts a new Nvim server using the same v:argv (except `-- [file…]` files and `-S [file]`), and sets v:startreason.
4. Restores the saved session (unless [!] was given).
5. Attaches all UIs to the new Nvim server and runs `[command]` on it.

Example: discard changes and stop with `:qall!`, then restart:

```
:restart +qall!
```

Example: restart and (manually) restore the session:

```
:mksession! Session.vim | restart! source Session.vim
```

Example: restart and update plugins:

```
:restart packupdate
```

Note:
- Only works if the UI and server are on the same system.
- Windows limitation: during +cmd execution, v:servername refers to a temporary address.
- If no UI handles the "restart" event, this leaves a dangling process.
- To adjust what `:restart` restores, set 'sessionoptions'.

Connect UI to a different server

:connect

:connect {address} Attaches the UI to another Nvim instance: detaches from the current server, then attaches to {address}.

Note: If the current UI hasn't implemented the "connect" UI event, this command is equivalent to :detach.

:connect! {address} Same as :connect but it also stops the detached server if no other UI is currently attached to it.

GUI commands

:winp :winpos E188 :winp[os] Display current position of the top left corner of the GUI vim window in pixels. Does not work in all versions. Also see getwinpos(), getwinposx() and getwinposy().

:winp[os] {X} {Y} E466
Put the GUI vim window at the given {X} and {Y} coordinates. The coordinates should specify the position in pixels of the top left corner of the window. When the GUI window has not been opened yet, the values are remembered until the window is opened. The position is adjusted to make the window fit on the screen (if possible).

:wi :win :winsize E465 :win[size] {width} {height} Set the window height to {width} by {height} characters. Obsolete, use ":set lines=11 columns=22".

Using the mouse mouse-using

mouse-mode-table mouse-overview Overview of what the mouse buttons do, when 'mousemodel' is "extend":

<S-LeftMouse> <A-RightMouse> <S-RightMouse> <RightDrag> <RightRelease> <LeftDrag> Normal Mode:

```
event         position     selection        change  action
               cursor                       window
--------------------------------------------------------------------------
<LeftMouse>     yes          end              yes
<C-LeftMouse>   no         no change          no     toggle multicursor
<S-LeftMouse>   yes        no change          yes    "*" (2)
<LeftDrag>      yes     start or extend (1)   no
<LeftRelease>   yes     start or extend (1)   no
<MiddleMouse>   yes       if not active       no     put
<MiddleMouse>   yes       if active           no     yank and put
<RightMouse>    yes     start or extend       yes
<A-RightMouse>  yes start or extend blockw.   yes
<S-RightMouse>  yes        no change          yes    "#" (2)
<C-RightMouse>  no         no change          no     "CTRL-T"
<RightDrag>     yes         extend            no
<RightRelease>  yes         extend            no
```

Insert or Replace Mode:

```
event         position     selection        change  action
               cursor                       window
--------------------------------------------------------------------------
<LeftMouse>     yes     (cannot be active)    yes
<C-LeftMouse>   no      (cannot be active)    no     no-op
<S-LeftMouse>   yes     (cannot be active)    yes    "CTRL-O*" (2)
<LeftDrag>      yes     start or extend (1)   no     like CTRL-O (1)
<LeftRelease>   yes     start or extend (1)   no     like CTRL-O (1)
<MiddleMouse>   no      (cannot be active)    no     put register
<RightMouse>    yes     start or extend       yes    like CTRL-O
<A-RightMouse>  yes start or extend blockw.   yes
<S-RightMouse>  yes     (cannot be active)    yes    "CTRL-O#" (2)
<C-RightMouse>  no      (cannot be active)    no     "CTRL-O CTRL-T"
```

In a help window:

```
event         position     selection        change  action
               cursor                       window
--------------------------------------------------------------------------
<2-LeftMouse>   yes     (cannot be active)    no     "^]" (jump to help tag)
```

When 'mousemodel' is "popup", these are different:

<A-LeftMouse>

Normal Mode:

```
event         position     selection        change  action
               cursor                       window
-------------------------------------------------------------------------
<S-LeftMouse>   yes     start or extend (1)   no
<A-LeftMouse>   yes     start/extend blockw   no
<RightMouse>    no      popup menu            no
```

Insert or Replace Mode:

```
event         position     selection        change  action
               cursor                       window
-------------------------------------------------------------------------
<S-LeftMouse>   yes     start or extend (1)   no     like CTRL-O (1)
<A-LeftMouse>   yes     start/extend blockw   no
<RightMouse>    no      popup menu            no
```

(1) only if mouse pointer moved since press (2) only if click is in same buffer

Clicking the left mouse button causes the cursor to be positioned. If the click is in another window that window is made the active window. When editing the command-line the cursor can only be positioned on the command-line. When in Insert mode Vim remains in Insert mode. If 'scrolloff' is set, and the cursor is positioned within 'scrolloff' lines from the window border, the text is scrolled.

A selection can be started by pressing the left mouse button on the first character, moving the mouse to the last character, then releasing the mouse button. You will not always see the selection until you release the button, only in some versions (GUI, Win32) will the dragging be shown immediately. Note that you can make the text scroll by moving the mouse at least one character in the first/last line in the window when 'scrolloff' is non-zero.

In Normal, Visual and Select mode clicking the right mouse button causes the Visual area to be extended. When 'mousemodel' is "popup", the left button has to be used while keeping the shift key pressed. When clicking in a window which is editing another buffer, the Visual or Select mode is stopped.

In Normal, Visual and Select mode clicking the right mouse button with the alt key pressed causes the Visual area to become blockwise. When 'mousemodel' is "popup" the left button has to be used with the alt key. Note that this won't work on systems where the window manager consumes the mouse events when the alt key is pressed (it may move the window).

double-click <2-LeftMouse> <3-LeftMouse> <4-LeftMouse> Double, triple and quadruple clicks are supported. For selecting text, extra clicks extend the selection:

```
click           select
--------------------------------
double          word or % match
triple          line
quadruple       rectangular block
```

Exception: In a :help window, double-click jumps to help for the word that is clicked on.

Double-click on a word selects that word. 'iskeyword' is used to specify which characters are included in a word. Double-click on a character that has a match selects until that match (like using "v%"). If the match is an #if/#else/#endif block, the selection becomes linewise. The time for double-clicking can be set with the 'mousetime' option.

Example: configure double-click to jump to the tag under the cursor:

```
:map <2-LeftMouse> :exe "tag " .. expand("<cword>")<CR>
```

Dragging the mouse with a double-click (button-down, button-up, button-down and then drag) will result in whole words to be selected. This continues until the button is released, at which point the selection is per character again.

For scrolling with the mouse see scroll-mouse-wheel.

In Insert mode, when a selection is started, Vim goes into Normal mode temporarily. When Visual or Select mode ends, it returns to Insert mode. This is like using CTRL-O in Insert mode. Select mode is used when the 'selectmode' option contains "mouse".

X1Mouse X1Drag X1Release X2Mouse X2Drag X2Release <MiddleRelease> <MiddleDrag> Mouse clicks can be mapped using these keycodes:

```
    code           mouse button              normal action
-------------------------------------------------------------------------
<LeftMouse>     left pressed               set cursor position
<LeftDrag>      left moved while pressed   extend selection
<LeftRelease>   left released              set selection end
<MiddleMouse>   middle pressed             paste text at cursor position
<MiddleDrag>    middle moved while pressed -
<MiddleRelease> middle released            -
<RightMouse>    right pressed              extend selection
<RightDrag>     right moved while pressed  extend selection
<RightRelease>  right released             set selection end
<X1Mouse>       X1 button pressed          -
<X1Drag>        X1 moved while pressed     -
<X1Release>     X1 button release          -
<X2Mouse>       X2 button pressed          -
<X2Drag>        X2 moved while pressed     -
<X2Release>     X2 button release          -
```

The X1 and X2 buttons refer to the extra buttons found on some mice (e.g. the right thumb).

Examples:

```
:noremap <MiddleMouse> <LeftMouse><MiddleMouse>
```

Paste at the position of the middle mouse button click (otherwise the paste would be done at the cursor position).

```
:noremap <LeftRelease> <LeftRelease>y
```

Immediately yank the selection, when using Visual mode.

Note the use of ":noremap" instead of "map" to avoid a recursive mapping.

```
:map <X1Mouse> <C-O>
:map <X2Mouse> <C-I>
```

Map the X1 and X2 buttons to go forwards and backwards in the jump list, see CTRL-O and CTRL-I.

mouse-swap-buttons pi_swapmouse To swap the meaning of the left and right mouse buttons:

```
:noremap        <LeftMouse>     <RightMouse>
:noremap        <LeftDrag>      <RightDrag>
:noremap        <LeftRelease>   <RightRelease>
:noremap        <RightMouse>    <LeftMouse>
:noremap        <RightDrag>     <LeftDrag>
:noremap        <RightRelease>  <LeftRelease>
:noremap        g<LeftMouse>    <C-RightMouse>
:noremap        g<RightMouse>   <C-LeftMouse>
:noremap!       <LeftMouse>     <RightMouse>
:noremap!       <LeftDrag>      <RightDrag>
:noremap!       <LeftRelease>   <RightRelease>
:noremap!       <RightMouse>    <LeftMouse>
:noremap!       <RightDrag>     <LeftDrag>
:noremap!       <RightRelease>  <LeftRelease>
```

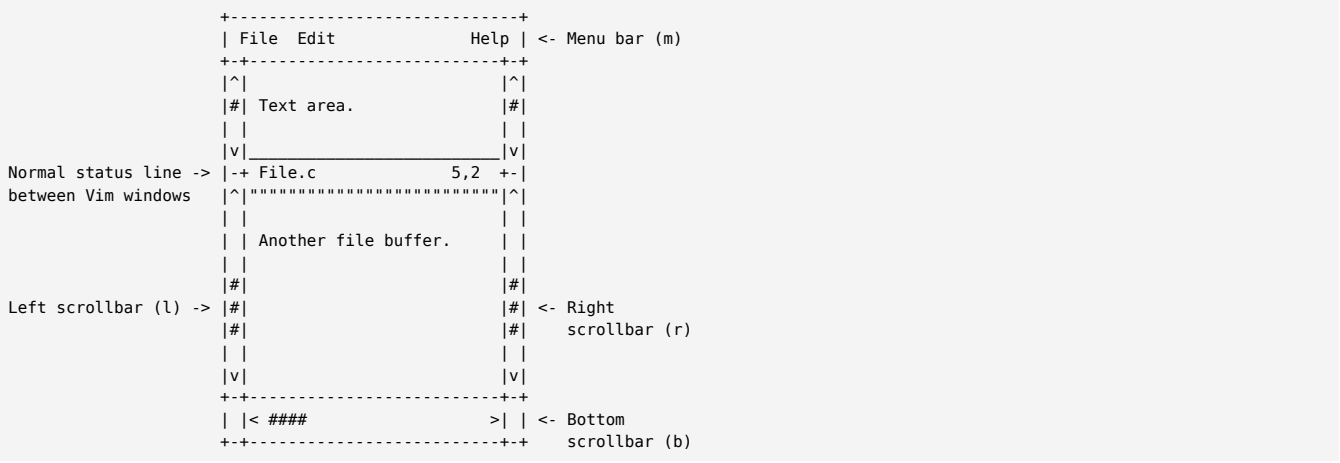
The swapmouse plugin does exactly this. Use pack-add to load it:

```
:packadd! swapmouse
```

Scrollbars gui-scrollbars

There are vertical scrollbars and a horizontal scrollbar. You may configure which ones appear with the 'guioptions' option.

The interface looks like this (with `:set guioptions=mlrb`):



Any of the scrollbar or menu components may be turned off by not putting the appropriate letter in the 'guioptions' string. The bottom scrollbar is only useful when 'nowrap' is set.

VERTICAL SCROLLBARS gui-vert-scroll

Each Vim window has a scrollbar next to it which may be scrolled up and down to move through the text in that buffer. The size of the scrollbar-thumb indicates the fraction of the buffer which can be seen in the window. When the scrollbar is dragged all the way down, the last line of the file will appear in the top of the window.

If a window is shrunk to zero height (by the growth of another window) its scrollbar disappears. It reappears when the window is restored.

If a window is vertically split, it will get a scrollbar when it is the current window and when, taking the middle of the current window and drawing a vertical line, this line goes through the window. When there are scrollbars on both sides, and the middle of the current window is on the left half, the right scrollbar column will contain scrollbars for the rightmost windows. The same happens on the other side.

HORIZONTAL SCROLLBARS gui-horiz-scroll

The horizontal scrollbar (at the bottom of the Vim GUI) may be used to scroll text sideways when the 'wrap' option is turned off. The scrollbar-thumb size is such that the text of the longest visible line may be scrolled as far as possible left and right. The cursor is moved when necessary, it must remain on a visible character (unless 'virtualedit' is set).

Computing the length of the longest visible line takes quite a bit of computation, and it has to be done every time something changes. If this takes too much time or you don't like the cursor jumping to another line, include the 'h' flag in 'guioptions'. Then the scrolling is limited by the text of the current cursor line.

Drag and drop

drag-n-drop

You can drag and drop one or more files into the Vim window, where they will be opened as if a :drop command was used.

If you hold down Shift while doing this, Vim changes to the first dropped file's directory. If you hold Ctrl Vim will always split a new window for the file. Otherwise it's only done if the current buffer has been changed.

You can also drop a directory on Vim. This starts the explorer plugin for that directory (assuming it was enabled, otherwise you'll get an error message). Keep Shift pressed to change to the directory instead.

If Vim happens to be editing a command line, the names of the dropped files and directories will be inserted at the cursor. This allows you to use these names with any Ex command. Special characters (space, tab, double quote and "|"; backslash on non-MS-Windows systems) will be escaped.

Menus

menus

For an introduction see usr_42.txt in the user manual.

Using Menus

using-menus

Basically, menus can be used just like mappings. You can define your own menus, as many as you like. Long-time Vim users won't use menus much. But the power is in adding your own menus and menu items. They are most useful for things that you can't remember what the key sequence was.

For creating menus in a different language, see :menutrans.

menu.vim

The default menus are read from the file "$VIMRUNTIME/menu.vim". See $VIMRUNTIME for where the path comes from. You can set up your own menus. Starting off with the default set is a good idea. You can add more items, or, if you don't like the defaults at all, start with removing all menus :unmenu-all. You can also avoid the default menus being loaded by adding this line to your vimrc file (NOT your gvimrc file!):

```
:let did_install_default_menus = 1
```

If you also want to avoid the Syntax menu:

```
:let did_install_syntax_menu = 1
```

The first item in the Syntax menu can be used to show all available filetypes in the menu (which can take a bit of time to load). If you want to have all filetypes already present at startup, add:

```
:let do_syntax_sel_menu = 1
```

Note that the menu.vim is sourced when `:syntax on` or `:filetype on` is executed or after your .vimrc file is sourced. This means that the 'encoding' option and the language of messages (`:language messages`) must be set before that (if you want to change them).

console-menus

Although this documentation is in the GUI section, you can actually use menus in console mode too. You will have to load menu.vim explicitly then, it is not done by default. You can use the :emenu command and command-line completion with 'wildmenu' to access the menu entries almost like a real menu system. To do this, put these commands in your vimrc file:

```
:source $VIMRUNTIME/menu.vim
:set wildmenu
:set cpo-=<
:set wcm=<C-Z>
:map <F4> :emenu <C-Z>
```

Pressing <F4> will start the menu. You can now use the cursor keys to select a menu entry. Hit <Enter> to execute it. Hit <Esc> if you want to cancel.

Creating New Menus

creating-menus

:me :menu :noreme :noremenu E330 E327 E331 E336 E333 E328 E329 E337 E792 E1310 To create a new menu item, use the ":menu" commands. They are mostly like the ":map" set of commands (see map-modes), but the first argument is a menu item name, given as a path of menus and submenus with a '.' between them, e.g.:

```
:menu File.Save  :w<CR>
:inoremenu File.Save  <C-O>:w<CR>
:menu Edit.Big\ Changes.Delete\ All\ Spaces  :%s/[ ^I]//g<CR>
```

This last one will create a new item in the menu bar called "Edit", holding the mouse button down on this will pop up a menu containing the item "Big Changes", which is a sub-menu containing the item "Delete All Spaces", which when selected, performs the operation.

To create a menu for terminal mode, use :tlmenu instead of :tmenu unlike key mapping (:tmap). This is because :tmenu is already used for defining tooltips for menus. See terminal-input.

Special characters in a menu name:

menu-shortcut

- & The next character is the shortcut key. Make sure each shortcut key is only used once in a (sub)menu. If you want to insert a literal "&" in the menu name use "&&". menu-text
- <Tab> Separates the menu name from right-aligned text. This can be used to show the equivalent typed command. The text "<Tab>" can be used here for convenience. If you are using a real tab, don't forget to put a backslash before it!

Example:

```
:amenu &File.&Open<Tab>:e  :browse e<CR>
```

[typed literally] With the shortcut "F" (while keeping the <Alt> key pressed), and then "O", this menu can be used. The second part is shown as "Open :e". The ":e" is right aligned, and the "O" is underlined, to indicate it is the shortcut.

:am :amenu :an :anoremenu The ":amenu" command can be used to define menu entries for all modes at once, except for Terminal mode. To make the command work correctly, a character is automatically inserted for some modes:

```
mode            inserted        appended
Normal          nothing         nothing
Visual          <C-C>           <C-\><C-G>
Insert          <C-\><C-O>
Cmdline         <C-C>           <C-\><C-G>
Op-pending      <C-C>           <C-\><C-G>
```

Example:

```
:amenu File.Next     :next^M
```

is equal to:

```
:nmenu File.Next     :next^M
:vmenu File.Next     ^C:next^M^\^G
:imenu File.Next     ^\^O:next^M
:cmenu File.Next     ^C:next^M^\^G
:omenu File.Next     ^C:next^M^\^G
```

Careful: In Insert mode this only works for a SINGLE Normal mode command, because of the CTRL-O. If you have two or more commands, you will need to use the ":imenu" command. For inserting text in any mode, you can use the expression register:

```
:amenu Insert.foobar   "='foobar'<CR>P
```

The special text <Cmd> begins a "command menu", it executes the command directly without changing modes. Where you might use ":...<CR>" you can instead use "<Cmd>...<CR>". See <Cmd> for more info. Example:

```
anoremenu File.Next <Cmd>next<CR>
```

Note that <Esc> in Cmdline mode executes the command, like in a mapping. This is Vi compatible. Use CTRL-C to quit Cmdline mode.

:nme :nmenu :nnoreme :nnoremenu :nunme :nunmenu Menu commands starting with "n" work in Normal mode. mapmode-n

:ome :omenu :onoreme :onoremenu :ounme :ounmenu Menu commands starting with "o" work in Operator-pending mode. mapmode-o

:vme :vmenu :vnoreme :vnoremenu :vunme :vunmenu Menu commands starting with "v" work in Visual mode. mapmode-v

:xme :xmenu :xnoreme :xnoremenu :xunme :xunmenu Menu commands starting with "x" work in Visual and Select mode. mapmode-x

:sme :smenu :snoreme :snoremenu :sunme :sunmenu Menu commands starting with "s" work in Select mode. mapmode-s

:ime :imenu :inoreme :inoremenu :iunme :iunmenu Menu commands starting with "i" work in Insert mode. mapmode-i

:cme :cmenu :cnoreme :cnoremenu :cunme :cunmenu Menu commands starting with "c" work in Cmdline mode. mapmode-c

:tlm :tlmenu :tln :tlnoremenu :tlu :tlunmenu Menu commands starting with "tl" work in Terminal mode. mapmode-t

:menu-<silent> :menu-silent To define a menu which will not be echoed on the command line, add "<silent>" as the first argument. Example:

```
:menu <silent> Settings.Ignore\ case  :set ic<CR>
```

The ":set ic" will not be echoed when using this menu. Messages from the executed command are still given though. To shut them up too, add a ":silent" in the executed command:

```
:menu <silent> Search.Header :exe ":silent normal /Header\r"<CR>
```

"<silent>" may also appear just after "<script>".

:menu-<script> :menu-script The "to" part of the menu will be inspected for mappings. If you don't want this, use the ":noremenu" command (or the similar one for a specific mode). If you do want to use script-local mappings, add "<script>" as the very first argument to the ":menu" command or just after "<silent>".

menu-priority

You can give a priority to a menu. Menus with a higher priority go more to the right. The priority is given as a number before the ":menu" command. Example:

```
:80menu Buffer.next :bn<CR>
```

The default menus have these priorities:

```
File            10
Edit            20
Tools           40
Syntax          50
Buffers         60
Window          70
Help            9999
```

When no or zero priority is given, 500 is used. The priority for the PopUp menu is not used.

You can use a priority higher than 9999, to make it go after the Help menu, but that is non-standard and is discouraged. The highest possible priority is about 32000. The lowest is 1.

sub-menu-priority

The same mechanism can be used to position a sub-menu. The priority is then given as a dot-separated list of priorities, before the menu

name:

```
:menu 80.500 Buffer.next :bn<CR>
```

Giving the sub-menu priority is only needed when the item is not to be put in a normal position. For example, to put a sub-menu before the other items:

```
:menu 80.100 Buffer.first :brew<CR>
```

Or to put a sub-menu after the other items, and further items with default priority will be put before it:

```
:menu 80.900 Buffer.last :blast<CR>
```

When a number is missing, the default value 500 will be used:

```
:menu .900 myMenu.test :echo "text"<CR>
```

The menu priority is only used when creating a new menu. When it already existed, e.g., in another mode, the priority will not change. Thus, the priority only needs to be given the first time a menu is used. An exception is the PopUp menu. There is a separate menu for each mode (Normal, Op-pending, Visual, Insert, Cmdline). The order in each of these menus can be different. This is different from menu-bar menus, which have the same order for all modes. **NOTE:** sub-menu priorities currently don't work for all versions of the GUI.

menu-separator E332 Menu items can be separated by a special item that inserts some space between items. Depending on the system this is displayed as a line or a dotted line. These items must start with a '-' and end in a '-'. The part in between is used to give it a unique name. Priorities can be used as with normal items. Example:

```
:menu Example.item1     :do something
:menu Example.-Sep-     :
:menu Example.item2     :do something different
```

Note that the separator also requires a rhs. It doesn't matter what it is, because the item will never be selected. Use a single colon to keep it simple.

gui-toolbar

The default toolbar is setup in menu.vim. The display of the toolbar is controlled by the 'guioptions' letter 'T'. You can thus have menu & toolbar together, or either on its own, or neither. The appearance is controlled by the 'toolbar' option. You can choose between an image, text or both.

toolbar-icon

The toolbar is defined as a special menu called ToolBar, which only has one level. Vim interprets the items in this menu as follows:

- 1 If an "icon=" argument was specified, the file with this name is used. The file can either be specified with the full path or with the base name. In the last case it is searched for in the "bitmaps" directory in 'runtimepath', like in point 3. Examples:

  ```
  :amenu icon=/usr/local/pixmaps/foo_icon.xpm ToolBar.Foo :echo "Foo"<CR>
  :amenu icon=FooIcon ToolBar.Foo :echo "Foo"<CR>
  ```

  Note that in the first case the extension is included, while in the second case it is omitted. If the file cannot be opened the next points are tried. A space in the file name must be escaped with a backslash. A menu priority must come _after_ the icon argument:

  ```
  :amenu icon=foo 1.42 ToolBar.Foo :echo "42!"<CR>
  ```

- 2 An item called 'BuiltIn##', where ## is a number, is taken as number ## of the built-in bitmaps available in Vim. Currently there are 31 numbered from 0 to 30 which cover most common editing operations builtin-tools.

  ```
  :amenu ToolBar.BuiltIn22 :call SearchNext("back")<CR>
  ```

- 3 An item with another name is first searched for in the directory "bitmaps" in 'runtimepath'. If found, the bitmap file is used as the toolbar button image. Note that the exact filename is OS-specific: For example, under Win32 the command

  ```
  :amenu ToolBar.Hello :echo "hello"<CR>
  ```

  would find the file 'hello.bmp'. Under X11 it is 'Hello.xpm'. For MS-Windows and the bitmap is scaled to fit the button. For MS-Windows a size of 18 by 18 pixels works best. For MS-Windows the bitmap should have 16 colors with the standard palette. The light grey pixels will be changed to the Window frame color and the dark grey pixels to the window shadow color. More colors might also work, depending on your system.
- 4 If the bitmap is still not found, Vim checks for a match against its list of built-in names. Each built-in button image has a name. So the command

  ```
  :amenu ToolBar.Open :e
  ```

  will show the built-in "open a file" button image if no open.bmp exists. All the built-in names can be seen used in menu.vim.
- 5 If all else fails, a blank, but functioning, button is displayed.

builtin-tools

```
nr  Name               Normal action
00  New                open new window
01  Open               browse for file to open in current window
02  Save               write buffer to file
03  Undo               undo last change
04  Redo               redo last undone change
05  Cut                delete selected text to clipboard
06  Copy               copy selected text to clipboard
07  Paste              paste text from clipboard
08  Print              print current buffer
09  Help               open a buffer on Vim's builtin help
10  Find               start a search command
11  SaveAll            write all modified buffers to file
12  SaveSesn           write session file for current situation
13  NewSesn            write new session file
14  LoadSesn           load session file
15  RunScript          browse for file to run as a Vim script
16  Replace            prompt for substitute command
17  WinClose           close current window
18  WinMax             make current window use many lines
19  WinMin             make current window use few lines
20  WinSplit           split current window
21  Shell              start a shell
22  FindPrev           search again, backward
23  FindNext           search again, forward
24  FindHelp           prompt for word to search help for
25  Make               run make and jump to first error
26  TagJump            jump to tag under the cursor
27  RunCtags           build tags for files in current directory
28  WinVSplit          split current window vertically
29  WinMaxWidth        make current window use many columns
30  WinMinWidth        make current window use few columns
```

hidden-menus win32-hidden-menus In the Win32 GUI, starting a menu name with ']' excludes that menu from the main menu bar. You must then use the :popup command to display it.

When splitting the window the window toolbar is not copied to the new window.

popup-menu

You can define the special menu "PopUp". This is the menu that is displayed when the right mouse button is pressed, if 'mousemodel' is set to popup or popup_setpos.

The default "PopUp" menu is:

```
amenu     PopUp.Open\ in\ web\ browser  gx
anoremenu PopUp.Inspect                 <Cmd>Inspect<CR>
anoremenu PopUp.Go\ to\ definition      <Cmd>lua vim.lsp.buf.definition()<CR>
anoremenu PopUp.Show\ Diagnostics       <Cmd>lua vim.diagnostic.open_float()<CR>
anoremenu PopUp.Show\ All\ Diagnostics  <Cmd>lua vim.diagnostic.setqflist()<CR>
anoremenu PopUp.Configure\ Diagnostics  <Cmd>help vim.diagnostic.config()<CR>
anoremenu PopUp.-1-                     <Nop>
vnoremenu PopUp.Cut                     "+x
vnoremenu PopUp.Copy                    "+y
anoremenu PopUp.Paste                   "+gP
vnoremenu PopUp.Paste                   "+P
vnoremenu PopUp.Delete                  "_x
nnoremenu PopUp.Select\ All             ggVG
vnoremenu PopUp.Select\ All             gg0oG$
inoremenu PopUp.Select\ All             <C-Home><C-O>VG
anoremenu PopUp.-2-                     <Nop>
anoremenu PopUp.How-to\ disable\ mouse  <Cmd>help disable-mouse<CR>
```

Showing What Menus Are Mapped To showing-menus

To see what an existing menu is mapped to, use just one argument after the menu commands (just like you would with the ":map" commands). If the menu specified is a submenu, then all menus under that hierarchy will be shown. If no argument is given after :menu at all, then ALL menu items are shown for the appropriate mode (e.g., Command-line mode for :cmenu).

Special characters in the list, just before the rhs:
- * Menu was defined with "nore" to disallow remapping.
- & Menu was defined with "<script>" to allow remapping script-local mappings.
- s Menu was defined with "<silent>" to avoid showing what it is mapped to when triggered.
- - Menu was disabled.

Note that hitting <Tab> while entering a menu name after a menu command may be used to complete the name of the menu item.

Executing Menus execute-menus

:em :emenu E334 E335 :[range]em[enu] {menu} Execute {menu} from the command line. The default is to execute the Normal mode menu. If a range is specified, it executes the Visual mode menu. If used from <c-o>, it executes the insert-mode menu Eg:

```
:emenu File.Exit
```

:[range]em[enu] {mode} {menu} Like above, but execute the menu for {mode}:
- 'n': :nmenu Normal mode
- 'v': :vmenu Visual mode
- 's': :smenu Select mode
- 'o': :omenu Operator-pending mode
- 't': :tlmenu Terminal mode

- 'i': :imenu Insert mode
- 'c': :cmenu Cmdline mode

You can use :emenu to access useful menu items you may have got used to from GUI mode. See 'wildmenu' for an option that works well with this. See console-menus for an example.

When using a range, if the lines match with '<,'>, then the menu is executed using the last visual selection.

Deleting Menus delete-menus

:unme :unmenu :aun :aunmenu To delete a menu item or a whole submenu, use the unmenu commands, which are analogous to the unmap commands. Eg:

```
:unmenu! Edit.Paste
```

This will remove the Paste item from the Edit menu for Insert and Command-line modes.

Note that hitting <Tab> while entering a menu name after an umenu command may be used to complete the name of the menu item for the appropriate mode.

To remove all menus use: :unmenu-all

```
:unmenu *       " remove all menus in Normal and visual mode
:unmenu! *      " remove all menus in Insert and Command-line mode
:aunmenu *      " remove all menus in all modes, except for Terminal
                " mode
:tlunmenu *     " remove all menus in Terminal mode
```

If you want to get rid of the menu bar:

```
:set guioptions-=m
```

Disabling Menus disable-menus

:menu-disable :menu-enable If you do not want to remove a menu, but disable it for a moment, this can be done by adding the "enable" or "disable" keyword to a ":menu" command. Examples:

```
:menu disable &File.&Open\.\.\.
:amenu enable *
:amenu disable &Tools.*
```

The command applies to the modes as used with all menu commands. Note that characters like "&" need to be included for translated names to be found. When the argument is "*", all menus are affected. Otherwise the given menu name and all existing submenus below it are affected.

Examples for Menus menu-examples

Here is an example on how to add menu items with menus! You can add a menu item for the keyword under the cursor. The register "z" is used.

```
:nmenu Words.Add\ Var         wb"zye:menu! Words.<C-R>z <C-R>z<CR>
:nmenu Words.Remove\ Var      wb"zye:unmenu! Words.<C-R>z<CR>
:vmenu Words.Add\ Var         "zy:menu! Words.<C-R>z <C-R>z <CR>
:vmenu Words.Remove\ Var      "zy:unmenu! Words.<C-R>z<CR>
:imenu Words.Add\ Var         <Esc>wb"zye:menu! Words.<C-R>z <C-R>z<CR>a
:imenu Words.Remove\ Var      <Esc>wb"zye:unmenu! Words.<C-R>z<CR>a
```

(the rhs is in <> notation, you can copy/paste this text to try out the mappings, or put these lines in your gvimrc; "<C-R>" is CTRL-R, "<CR>" is the <CR> key. <>)

tooltips menu-tips Tooltips & Menu tips

See section 42.4 in the user manual.

:tmenu

:tm[enu] {menupath} {rhs} Define a tip for a menu or tool. (only in X11 and Win32 GUI)

:tm[enu] [menupath] List menu tips. (only in X11 and Win32 GUI)

:tunmenu

:tu[nmenu] {menupath} Remove a tip for a menu or tool. (only in X11 and Win32 GUI)

Note: To create menus for terminal mode, use :tlmenu instead.

When a tip is defined for a menu item, it appears in the command-line area when the mouse is over that item, much like a standard Windows menu hint in the status bar. (Except when Vim is in Command-line mode, when of course nothing is displayed.) When a tip is defined for a ToolBar item, it appears as a tooltip when the mouse pauses over that button, in the usual fashion. Use the hl-Tooltip highlight group to change its colors.

A "tip" can be defined for each menu item. For example, when defining a menu item like this:

```
:amenu MyMenu.Hello :echo "Hello"<CR>
```

The tip is defined like this:

```
:tmenu MyMenu.Hello Displays a greeting.
```

And delete it with:

```
:tunmenu MyMenu.Hello
```

Tooltips are currently only supported for the X11 and Win32 GUI. However, they should appear for the other gui platforms in the not too distant future.

The ":tmenu" command works just like other menu commands, it uses the same arguments. ":tunmenu" deletes an existing menu tip, in the same way as the other unmenu commands.

If a menu item becomes invalid (i.e. its actions in all modes are deleted) Vim deletes the menu tip (and the item) for you. This means that :aunmenu deletes a menu item - you don't need to do a :tunmenu as well.

5.9 Popup Menus

You can cause a menu to popup at the cursor. This behaves similarly to the PopUp menus except that any menu tree can be popped up.

:popup :popu :popu[p] {name} Popup the menu {name}. The menu named must have at least one subentry, but need not appear on the menu-bar (see hidden-menus).

:popu[p]! {name} Like above, but use the position of the mouse pointer instead of the cursor.

Example:

```
:popup File
```

will make the "File" menu (if there is one) appear at the text cursor (mouse pointer if ! was used).

```
:amenu ]Toolbar.Make    :make<CR>
:popup ]Toolbar
```

This creates a popup menu that doesn't exist on the main menu-bar.

Note that a menu that starts with ']' will not be displayed.

Checkhealth

health

vim.health is a minimal framework to help users troubleshoot configuration and any other environment conditions that a plugin might care about. Nvim ships with healthchecks for configuration, performance, python support, ruby support, clipboard support, and more.

To run all healthchecks, use:

```
:checkhealth
```

Plugin authors are encouraged to write new healthchecks. health-dev

COMMANDS

health-commands

:che :checkhealth

:che[ckhealth] Run all healthchecks.

E5009

Nvim depends on $VIMRUNTIME, 'runtimepath' and 'packpath' to find the standard "runtime files" for syntax highlighting, filetype-specific behavior, and standard plugins (including :checkhealth). If the runtime files cannot be found then those features will not work.

:che[ckhealth] {plugins}
Run healthcheck(s) for one or more plugins. E.g. to run only the standard Nvim healthcheck:

```
:checkhealth vim.health
```

To run the healthchecks for the "foo" and "bar" plugins (assuming they are on 'runtimepath' and they have implemented the Lua require("foo.health").check() interface):

```
:checkhealth foo bar
```

To run healthchecks for Lua submodules, use dot notation or "*" to refer to all submodules. For example Nvim provides vim.lsp and vim.treesitter:

```
:checkhealth vim.lsp vim.treesitter
:checkhealth vim*
```

USAGE

health-usage

Local mappings in the healthcheck buffer:

q Closes the window.

Global configuration:

g:health

g:health Dictionary with the following optional keys:

- style ('float'|nil) Set to "float" to display :checkhealth in

  a floating window instead of the default behavior.

  Example:

```
vim.g.health = { style = 'float' }
```

Local configuration:

Checkhealth sets its buffer filetype to "checkhealth". You can customize the buffer by handling the FileType event. For example if you don't want emojis in the health report:

```
autocmd FileType checkhealth :set modifiable | silent! %s/\v( ?[^\x00-\x7F])//g
```

Sometimes you might need to suppress filetype warnings if you have specified them dynamically. For example with the yaml.ansible filetype:

```
autocmd FileType checkhealth :set modifiable | silent! g/Unknown filetype 'yaml\.ansible'/d
```

Create a healthcheck

health-dev

Healthchecks are functions that check the user environment, configuration, or any other prerequisites that a plugin cares about. Nvim ships with healthchecks in:

- $VIMRUNTIME/autoload/health/
- $VIMRUNTIME/lua/vim/lsp/health.lua
- $VIMRUNTIME/lua/vim/treesitter/health.lua

- and more...

To add a new healthcheck for your own plugin, simply create a "health.lua" module on 'runtimepath' that returns a table with a "check()" function. Then :checkhealth will automatically find and invoke the function.

For example if your plugin is named "foo", define your healthcheck module at one of these locations (on 'runtimepath'):

- lua/foo/health/init.lua
- lua/foo/health.lua

If your plugin also provides a submodule named "bar" for which you want a separate healthcheck, define the healthcheck at one of these locations:

- lua/foo/bar/health/init.lua
- lua/foo/bar/health.lua

All such health modules must return a Lua table containing a check() function.

Copy this sample code into lua/foo/health.lua, replacing "foo" in the path with your plugin name:

```
local M = {}

M.check = function()
  vim.health.start("foo report")
  -- make sure setup function parameters are ok
  if check_setup() then
    vim.health.ok("Setup is correct")
  else
    vim.health.error("Setup is incorrect")
  end
  -- do some more checking
  -- ...
end

return M
```

error({msg}, {...}) vim.health.error()

Reports an error.

Parameters:

- {msg} (string)
- {...} (string|string[]) Optional advice

info({msg}) vim.health.info()

Reports an informational message.

Parameters:

- {msg} (string)

ok({msg}) vim.health.ok()

Reports a "success" message.

Parameters:

- {msg} (string)

start({name}) vim.health.start()

Starts a new report. Most plugins should call this only once, but if you want different sections to appear in your report, call this once per section.

Parameters:

- {name} (string)

warn({msg}, {...}) vim.health.warn()

Reports a warning.

Parameters:

- {msg} (string)

- {...} (string|string[]) Optional advice

Help on help files

Help tag conventions

help-context

Get specific help: It is possible to go directly to whatever you want help on, by giving an argument to the :help command. Prepend something to specify the context:

```
    WHAT                      PREPEND    EXAMPLE
Normal mode command               :help x
Visual mode command       v_      :help v_u
Insert mode command       i_      :help i_<Esc>
Command-line command      :       :help :quit
Command-line editing      c_      :help c_<Del>
Vim command argument      -       :help -r
Option                    '       :help 'textwidth'
Regular expression        /       :help /[
```

See help-summary for more contexts and an explanation.
See notation for an explanation of the help syntax.

SEARCH FOR HELP

Type ":help word", then hit CTRL-D to see matching help entries for "word". Or use ":helpgrep word". :helpgrep

Notation

notation

When syntax highlighting is used to read this, text that is not typed literally is often highlighted with the Special group. These are items in [], {} and <>, and CTRL-X.

Note that Vim uses all possible characters in commands. Sometimes the [], {} and <> are part of what you type, the context should make this clear.

- [] Characters in square brackets are optional.

  count [count]

- [count] An optional number that may precede the command to multiply

  or iterate the command. If no number is given, a count of one is used, unless otherwise noted. Note that in this manual the [count] is not mentioned in the description of the command, but only in the explanation. This was done to make the commands easier to look up. If the 'showcmd' option is on, the (partially) entered count is shown at the bottom of the window. You can use <Del> to erase the last digit (N<Del>).

  [quotex]

- ["x] An optional register designation where text can be stored.

  See registers. The x is a single character between 'a' and 'z' or 'A' and 'Z' or '"', and in some cases (with the put command) between '0' and '9', '%', '#', or others. The uppercase and lowercase letter designate the same register, but the lowercase letter is used to overwrite the previous register contents, while the uppercase letter is used to append to the previous register contents. Without the ""x" or with """" the stored text is put into the unnamed register.

  {}

- {} Curly braces denote parts of the command which must appear,

  but which can take a number of different values. The differences between Vim and Vi are also given in curly braces (this will be clear from the context).

  {char1-char2}

- {char1-char2} A single character from the range char1 to char2. For

  example: {a-z} is a lowercase letter. Multiple ranges may be concatenated. For example, {a-zA-Z0-9} is any alphanumeric character.

  {motion} movement

- {motion} A command that moves the cursor. These are explained in

  motion.txt.

  - Examples:

- w to start of next word
- b to begin of current word
- 4j four lines down
- /The<CR> to next occurrence of "The"

- This is used after an operator command to move over the

  text that is to be operated upon.
- If the motion includes a count and the operator also has

  a count, the two counts are multiplied. For example:
  "2d3w" deletes six words.
- The motion can be backwards, e.g. "db" to delete to the

  start of the word.
- The motion can also be a mouse click. The mouse is not

  supported in every terminal though.
- The ":omap" command can be used to map characters while an

  operator is pending.
- Ex commands can be used to move the cursor. This can be

  used to call a function that does some complicated motion.
  The motion is always charwise exclusive, no matter what
  ":" command is used. This means it's impossible to
  include the last character of a line without the line
  break (unless 'virtualedit' is set). If the Ex command
  changes the text before where the operator starts or jumps
  to another buffer the result is unpredictable. It is
  possible to change the text further down. Jumping to
  another buffer is possible if the current buffer is not
  unloaded.

{Visual}

- {Visual} A selected text area. It is started with the "v", "V", or

  CTRL-V command, then any cursor movement command can be used
  to change the end of the selected text.
  This is used before an operator command to highlight the
  text that is to be operated upon.
  See Visual-mode.

<character>

- <character> A special character from the table below, optionally with

  modifiers, or a single ASCII character with modifiers.

'character'

- 'c' A single ASCII character.

CTRL-{char}

- CTRL-{char} {char} typed as a control character; that is, typing {char}

  while holding the CTRL key down. The case of {char} is
  ignored; thus CTRL-A and CTRL-a are equivalent. But in
  some terminals and environments, using the SHIFT key will
  produce a distinct code (e.g. CTRL-SHIFT-a); in these
  environments using the SHIFT key will not trigger commands
  such as CTRL-A.

'option'

- 'option' An option, or parameter, that can be set to a value, is

  enclosed in single quotes. See options.

quotecommandquote

- "command" A reference to a command that you can type is enclosed in double quotes.
- command New style command, this distinguishes it from other quoted text and strings.

Help commands

online-help

help <Help> :h :help <F1> i_<F1> i_<Help>

<Help> or :h[elp] Open a window and display the help file in read-only mode. If there is a help window open already, use that one. Otherwise, if the current window uses the full width of the screen or is at least 80 characters wide, the help window will appear just above the current window. Otherwise the new window is put at the very top.
The 'helplang' option is used to select a language, if the main help file is available in several languages.

:help!

:h[elp]! Guesses a help tag from the WORD at cursor, in DWIM ("Do What I Mean") fashion: trims punctuation using various heuristics until a valid help tag is found.

For example, move the cursor anywhere in this code, then run :help!:

```
local v = vim.version.parse(vim.system({'foo'}):wait().stdout)
```

Then compare to :exe 'help' expand('<cword>').

{subject} E149 E661

:h[elp] {subject} Like ":help", additionally jump to the tag {subject}. For example:

```
:help options
```

{subject} can include wildcards such as "*", "?" and "[a-z]":
   :help z? jump to help for any "z" command
   :help z. jump to the help for "z."
But when a tag exists it is taken literally:
   :help :? jump to help for ":?"

If there is no full match for the pattern, or there are several matches, the "best" match will be used. A sophisticated algorithm is used to decide which match is better than another one. These items are considered in the computation:

- A match with same case is much better than a match with different case.
- A match that starts after a non-alphanumeric character is better than a match in the middle of a word.
- A match at or near the beginning of the tag is better than a match further on.
- The more alphanumeric characters match, the better.
- The shorter the length of the match, the better.

The 'helplang' option is used to select a language, if the {subject} is available in several languages. To find a tag in a specific language, append "@ab", where "ab" is the two-letter language code. See help-translated.

Note that the longer the {subject} you give, the less matches will be found. You can get an idea how this all works by using commandline completion (type CTRL-D after ":help subject" c_CTRL-D). If there are several matches, you can have them listed by hitting CTRL-D. Example:

```
:help cont<Ctrl-D>
```

Instead of typing ":help CTRL-V" to search for help for CTRL-V you can type:

```
:help ^V
```

This also works together with other characters, for
example to find help for CTRL-V in Insert mode:

```
:help i^V
```

It is also possible to first do ":help" and then
use ":tag {pattern}" in the help window. The
":tnext" command can then be used to jump to other
matches, "tselect" to list matches and choose one.

```
:help index
:tselect /.*mode
```

When there is no argument you will see matches for
"help", to avoid listing all possible matches (that
would be very slow).
The number of matches displayed is limited to 300.

The :help command can be followed by '|' and another
command, but you don't need to escape the '|' inside a
help command. So these both work:

```
:help |
:help k| only
```

Note that a space before the '|' is seen as part of
the ":help" argument.
You can also use <NL> or <CR> to separate the help
command from a following command. You need to type
CTRL-V first to insert the <NL> or <CR>. Example:

```
:help so<C-V><CR>only
```

:h[elp]! {subject} Like ":help", but prefers tags with the same
help-translated language as the current file.

Unrelated to :help! (no {subject}).

:helpc :helpclose

:helpc[lose] Close one help window, if there is one.
Vim will try to restore the window layout (including
cursor position) to the same layout it was before
opening the help window initially. This might cause
triggering several autocommands.

:helpg :helpgrep

:helpg[rep] {pattern}[@xx]
Search all help text files and make a list of lines
in which {pattern} matches. Jumps to the first match.
The optional [@xx] specifies that only matches in the
"xx" language are to be found.
You can navigate through the matches with the
quickfix commands, e.g., :cnext to jump to the
next one. Or use :cwindow to get the list of
matches in the quickfix window.
{pattern} is used as a Vim regexp pattern.
'ignorecase' is not used, add "\c" to ignore case.
Example for case sensitive search:

```
:helpgrep Uganda
```

Example for case ignoring search:

```
:helpgrep uganda\c
```

Example for searching in French help:

```
:helpgrep backspace@fr
```

The pattern does not support line breaks, it must
match within one line. You can use :grep instead,
but then you need to get the list of help files in a
complicated way.
Cannot be followed by another command, everything is
used as part of the pattern. But you can use
:execute when needed.
Compressed help files will not be searched (Fedora
compresses the help files).

:lh :lhelpgrep

:lh[elpgrep] {pattern}[@xx]
Same as ":helpgrep", except the location list is used
instead of the quickfix list. If the help window is
already opened, then the location list for that window
is used. Otherwise, a new help window is opened and
the location list for that window is set. The
location list for the current window is not changed
then.

:exu :exusage

:exu[sage] Show help on Ex commands. Added to simulate the Nvi
command.

:viu :viusage

:viu[sage] Show help on Normal mode commands. Added to simulate the Nvi command.

When no argument is given to :help the file given with the 'helpfile' option will be opened. Otherwise the specified tag is searched for in all "doc/tags" files in the directories specified in the 'runtimepath' option.

If you would like to open the help in the current window, see this tip: help-curwin.

The initial height of the help window can be set with the 'helpheight' option (default 20).

help-buffer-options

When the help buffer is created, several local options are set to make sure the help text is displayed as it was intended:

'iskeyword'	nearly all ASCII chars except ' ', "*", '"' and '|'
'foldmethod'	"manual"
'tabstop'	8
'arabic'	off
'binary'	off
'buflisted'	off
'cursorbind'	off
'diff'	off
'foldenable'	off
'list'	off
'modifiable'	off
'number'	off
'relativenumber'	off
'rightleft'	off
'scrollbind'	off
'spell'	off

Jump to specific subjects by using tags. This can be done in two ways:

- Use the "CTRL-]" command while standing on the name of a command or option.

  This only works when the tag is a keyword. "g<LeftMouse>" works just like "CTRL-]".
- use the ":ta {subject}" command. This also works with non-keyword

  characters.

Use CTRL-T or CTRL-O to jump back.
Use ":q" to close the help window.
Use g== to execute the current Lua/Vimscript code block.

If there are several matches for an item you are looking for, this is how you can jump to each one of them:

1. Open a help window

2. Use the ":tag" command with a slash prepended to the tag. E.g.:

```
:tag /min
```

3. Use ":tnext" to jump to the next matching tag.

It is possible to add help files for plugins and other items. You don't need to change the distributed help files for that. See add-local-help.

To write a local help file, see write-local-help.

Note that the title lines from the local help files are automagically added to This is done when viewing the file in Vim, the file itself is not changed. It is done by going through all help files and obtaining the first line of each file. The files in $VIMRUNTIME/doc are skipped.

:helpt :helptags

E150 E151 E152 E153 E154 E670 E856

:helpt[ags] [++t] {dir}
Generate the help tags file(s) for directory {dir}. When {dir} is ALL then all "doc" directories in 'runtimepath' will be used.

All "*.txt" and "*.??x" files in the directory and sub-directories are scanned for a help tag definition in between stars. The "*.??x" files are for translated docs, they generate the "tags-??" file, see help-translated. The generated tags files are sorted.
When there are duplicates an error message is given. An existing tags file is silently overwritten.
Requires the "vimdoc" treesitter parser.

The optional "++t" argument forces adding the "help-tags" tag. This is also done when the {dir} is equal to $VIMRUNTIME/doc.

To rebuild the help tags in the runtime directory (requires write permission there):

```
:helptags $VIMRUNTIME/doc
```

Translated help files help-translated

It is possible to add translated help files, next to the original English help files. Vim will search for all help in "doc" directories in 'runtimepath'.

At this moment translations are available for:
```
        Chinese  - multiple authors
        French   - translated by David Blanchet
        Italian  - translated by Antonio Colombo
        Japanese - multiple authors
        Polish   - translated by Mikolaj Machowski
        Russian  - translated by Vassily Ragosin
```
See the Vim website to find them: https://www.vim.org/translations.php

A set of translated help files consists of these files:

```
        help.abx
        howto.abx
        ...
        tags-ab
```

"ab" is the two-letter language code. Thus for Italian the names are:

```
        help.itx
        howto.itx
        ...
        tags-it
```

The 'helplang' option can be set to the preferred language(s). The default is set according to the environment. Vim will first try to find a matching tag in the preferred language(s). English is used when it cannot be found.

To find a tag in a specific language, append "@ab" to a tag, where "ab" is the two-letter language code. Example:

```
:he user-manual@it
:he user-manual@en
```

The first one finds the Italian user manual, even when 'helplang' is empty. The second one finds the English user manual, even when 'helplang' is set to "it".

When using command-line completion for the ":help" command, the "@en" extension is only shown when a tag exists for multiple languages. When the tag only exists for English "@en" is omitted. When the first candidate has an "@ab" extension and it matches the first language in 'helplang' "@ab" is also omitted.

When using CTRL-] or ":help!" in a non-English help file Vim will try to find the tag in the same language. If not found then 'helplang' will be used to select a language.

Help files must use latin1 or utf-8 encoding. Vim assumes the encoding is utf-8 when finding non-ASCII characters in the first line. Thus you must translate the header with "For Vim version".

The same encoding must be used for the help files of one language in one directory. You can use a different encoding for different languages and use a different encoding for help files of the same language but in a different directory.

Hints for translators:
- Do not translate the tags. This makes it possible to use 'helplang' to specify the preferred language. You may add new tags in your language.
- When you do not translate a part of a file, add tags to the English version, using the "tag@en" notation.
- Make a package with all the files and the tags file available for download. Users can drop it in one of the "doc" directories and start use it. Report to the development team, so they can add a link on www.vim.org.
- Use the :helptags command to generate the tags files. It will find all languages in the specified directory.

Writing help files help-writing

For ease of use, a Vim help file for a plugin should follow the format of the standard Vim help files, except for the first line. If you are writing a new help file it's best to copy one of the existing files and use it as a template.

The first line in a help file should have the following format:

```
*plugin_name.txt*       {short description of the plugin}
```

The first field is a help tag where ":help plugin_name" will jump to. The remainder of the line, after a Tab, describes the plugin purpose in a short way. This will show up in the "LOCAL ADDITIONS" section of the main help

If you want to add a version number or last modification date, put it in the second line, right aligned.

At the bottom of the help file, place a Vim modeline to set the 'textwidth' and 'tabstop' options and the 'filetype' to "help". Never set a global option in such a modeline, that can have undesired consequences.

STYLE

If your Vim has 'modeline' enabled, Vim should follow the preferred style automatically when editing built-in help files.

Vim help files should be formatted for a 'textwidth' of 78 characters, so they look good in a typical 80 x 24 terminal window.

Use two spaces between the final dot of a sentence of the first letter of the next sentence. Like this.

Use tab characters for aligning content, with a 'tabstop' setting of 8. This also helps reduce the file size.

Always use :retab after you have finished editing. Don't blindly use :retab!, always review what will be changed to avoid unwanted changes.

TAGS

To define a help tag, place the name between asterisks ("*tag-name*"). The tag-name should be different from all the Vim help tag names and ideally should begin with the name of the Vim plugin. The tag name is usually right aligned on a line.

When referring to an existing help tag and to create a hot-link, place the name between two bars ("|") eg. help-writing.

When referring to a Vim option in the help file, place the option name between two single quotes, eg. 'statusline'

When referring to any other technical term or symbol, such as a filename or function parameter, surround it in backticks, eg. ~/.path/to/init.vim. This renders it as "inline" code (as opposed to a "codeblock" help-codeblock).

HIGHLIGHTING

To define a column heading, use a tilde character at the end of the line, preceded by a space. This will highlight the column heading in a different color. E.g.

Column heading

To separate sections in a help file, place a series of '=' characters in a line starting from the first column. The section separator line is highlighted differently.

help-codeblock

To quote a block of ex-commands verbatim, place a greater than (>) character at the end of the line before the block and a less than (<) character as the first non-blank on a line following the block. Any line starting in column 1 also implicitly stops the block of ex-commands before it. E.g.

```
function Example_Func()
  echo "Example"
endfunction
```

To enable syntax highlighting for a block of code, place a language name annotation (e.g. "vim") after a greater than (>) character. E.g.

```
function Example_Func()
  echo "Example"
endfunction
```

help-notation

The following are highlighted differently in a Vim help file:

- a special key name expressed either in <> notation as in <PageDown>, or

  as a Ctrl character as in CTRL-X
- anything between {braces}, e.g. {lhs} and {rhs}

The word "Note", "Notes" and similar automagically receive distinctive highlighting. So do these:

```
Todo    something to do
Error   something wrong
```

You can find the details in $VIMRUNTIME/syntax/help.vim

FILETYPE COMPLETION

ft-help-omni

To get completion for help tags when writing a tag reference, you can use the i_CTRL-X_CTRL-O command.

The perl Interface to Vim perl

See provider-perl for more information.

1. Commands perl-commands

:perl

:[range]perl {stmt}
Execute perl statement {stmt}. The current package is "main". A simple check if the :perl command is working:

```
:perl print "Hello"
```

:[range]perl << [trim] [{endmarker}]
{script}
{endmarker}
Execute perl script {script}.
The {endmarker} after {script} must NOT be preceded by any white space.

If [endmarker] is omitted, it defaults to a dot '.' like for the :append and :insert commands.

Useful for including perl code in Vim scripts. Requires perl, see script-here.

Example:

```
function! MyVimMethod()
perl << EOF
sub my_vim_method
{
        print "Hello World!\n";
}
EOF
endfunction
```

To see what version of perl you have:

```
:perl print $^V
```

:perldo

:[range]perldo {cmd} Execute perl command {cmd} for each line in the[range], with $_ being set to the test of each line in turn, without a trailing <EOL>. In addition to $_, $line and $linenr is also set to the line content and line number respectively. Setting $_ will change the text, but note that it is not possible to add or delete lines using this command.
The default for [range] is the whole file: "1,$".

Examples:

```
:perldo $_ = reverse($_);
:perldo $_ = "".$linenr." => $line";
```

One can use :perldo in conjunction with :perl to filter a range using perl. For example:

```
:perl << EOF
sub perl_vim_string_replace
{
    my $line = shift;
    my $needle = $vim->eval('@a');
    my $replacement = $vim->eval('@b');
    $line =~ s/$needle/$replacement/g;
    return $line;
}
EOF
:let @a='somevalue'
:let @b='newvalue'
:'<,'>perldo $_ = perl_vim_string_replace($_)
```

:perlfile

:[range]perlfile {file}
Execute the perl script in {file}. The whole argument is used as a single file name.

Both of these commands do essentially the same thing - they execute a piece of perl code, with the "current range" set to the given line range.

In the case of :perl, the code to execute is in the command-line.
In the case of :perlfile, the code to execute is the contents of the given file.

perl commands cannot be used in the sandbox.

To pass arguments you need to set @ARGV explicitly. Example:

```
:perl @ARGV = ("foo", "bar");
:perlfile myscript.pl
```

Here are some examples perl-examples

```
:perl print "Hello"
:perl $current->line (uc ($current->line))
:perl my $str = $current->buffer->[42]; print "Set \$str to: $str"
```

Note that changes (such as the "use" statements) persist from one command
to the next.

2. The VIM module perl-vim

Perl code gets all of its access to Nvim via the "VIM" module.

Overview

```
print "Hello"                          # displays a message
VIM::Msg("Hello")                      # displays a message
VIM::SetOption("ai")                   # sets a vim option
$nbuf = VIM::Buffers()                 # returns the number of buffers
@buflist = VIM::Buffers()              # returns array of all buffers
$mybuf = (VIM::Buffers('a.c'))[0]      # returns buffer object for 'a.c'
@winlist = VIM::Windows()              # returns array of all windows
$nwin = VIM::Windows()                 # returns the number of windows
($success, $v) = VIM::Eval('&path')    # $v: option 'path', $success: 1
($success, $v) = VIM::Eval('&xyz')     # $v: '' and $success: 0
$v = VIM::Eval('expand("<cfile>")')    # expands <cfile>
$curwin->SetHeight(10)                 # sets the window height
@pos = $curwin->Cursor()               # returns (row, col) array
@pos = (10, 10)
$curwin->Cursor(@pos)                  # sets cursor to @pos
$curwin->Cursor(10,10)                 # sets cursor to row 10 col 10
$mybuf = $curwin->Buffer()             # returns the buffer object for window
$curbuf->Name()                        # returns buffer name
$curbuf->Number()                      # returns buffer number
$curbuf->Count()                       # returns the number of lines
$l = $curbuf->Get(10)                  # returns line 10
@l = $curbuf->Get(1 .. 5)              # returns lines 1 through 5
$curbuf->Delete(10)                    # deletes line 10
$curbuf->Delete(10, 20)                # delete lines 10 through 20
$curbuf->Append(10, "Line")            # appends a line
$curbuf->Append(10, "L1", "L2", "L3")  # appends 3 lines
@l = ("L1", "L2", "L3")
$curbuf->Append(10, @l)                # appends L1, L2 and L3
$curbuf->Set(10, "Line")               # replaces line 10
$curbuf->Set(10, "Line1", "Line2")     # replaces lines 10 and 11
$curbuf->Set(10, @l)                   # replaces 3 lines
```

Module Functions:

perl-Msg

VIM::Msg({msg})
Displays the message {msg}.

perl-SetOption

VIM::SetOption({arg}) Sets a vim option. {arg} can be any argument that the
":set" command accepts. Note that this means that no
spaces are allowed in the argument! See :set.

perl-Buffers

VIM::Buffers([{bn}...]) With no arguments, returns a list of all the buffers
in an array context or returns the number of buffers
in a scalar context. For a list of buffer names or
numbers {bn}, returns a list of the buffers matching
{bn}, using the same rules as Vim's internal
bufname() function.
WARNING: the list becomes invalid when :bwipe is
used.

perl-Windows

VIM::Windows([{wn}...]) With no arguments, returns a list of all the windows
in an array context or returns the number of windows
in a scalar context. For a list of window numbers
{wn}, returns a list of the windows with those
numbers.
WARNING: the list becomes invalid when a window is
closed.

perl-DoCommand

VIM::DoCommand({cmd}) Executes Ex command {cmd}.

perl-Eval

VIM::Eval({expr}) Evaluates {expr} and returns (success, value) in list
context or just value in scalar context.
success=1 indicates that val contains the value of
{expr}; success=0 indicates a failure to evaluate
the expression. '@x' returns the contents of register
x, '&x' returns the value of option x, 'x' returns the
value of internal variables x, and '$x' is equivalent
to perl's $ENV{x}. All functions accessible from
the command-line are valid for {expr}.
A List is turned into a string by joining the items
and inserting line breaks.

perl-Blob

VIM::Blob({expr}) Return Blob literal string 0zXXXX from scalar value.

3. VIM::Buffer objects perl-buffer

Methods:

perl-Buffer-Name

Name() Returns the filename for the Buffer.

perl-Buffer-Number

Number() Returns the number of the Buffer.

perl-Buffer-Count

Count() Returns the number of lines in the Buffer.

perl-Buffer-Get

Get({lnum}, {lnum}?, ...)
Returns a text string of line {lnum} in the Buffer
for each {lnum} specified. An array can be passed
with a list of {lnum}'s specified.

perl-Buffer-Delete

Delete({lnum}, {lnum}?)
Deletes line {lnum} in the Buffer. With the second
{lnum}, deletes the range of lines from the first
{lnum} to the second {lnum}.

perl-Buffer-Append

Append({lnum}, {line}, {line}?, ...)
Appends each {line} string after Buffer line {lnum}.
The list of {line}s can be an array.

perl-Buffer-Set

Set({lnum}, {line}, {line}?, ...)
Replaces one or more Buffer lines with specified
{lines}s, starting at Buffer line {lnum}. The list of
{line}s can be an array. If the arguments are
invalid, replacement does not occur.

4. VIM::Window objects perl-window

Methods:

perl-Window-SetHeight

SetHeight({height})
Sets the Window height to {height}, within screen
limits.

perl-Window-GetCursor

Cursor({row}?, {col}?)
With no arguments, returns a (row, col) array for the
current cursor position in the Window. With {row} and
{col} arguments, sets the Window's cursor position to
{row} and {col}. Note that {col} is numbered from 0,
Perl-fashion, and thus is one less than the value in
Vim's ruler.

Buffer() perl-Window-Buffer

Returns the Buffer object corresponding to the given
Window.

5. Lexical variables perl-globals

There are multiple lexical variables.

$curwin The current Window object.
$curbuf The current Buffer object.
$vim A Neovim::Ext object.
$nvim The same as $nvim.
$current A Neovim::Ext::Current object.

These are also available via the "main" package:

```
$main::curwin           The current Window object.
$main::curbuf           The current Buffer object.
```

The Python Interface to NVim python Python

See provider-python for more information.

Commands

python-commands

:python :py E263 E264 E887

:[range]py[thon] {stmt}
Execute Python statement {stmt}. A simple check if the :python command is working:

```
:python print("Hello")
```

:[range]py[thon] << [trim] [{endmarker}]
{script}
{endmarker}
Execute Python script {script}. Useful for including python code in Vim scripts. Requires Python, see script-here.

If [endmarker] is omitted from after the "<<", a dot '.' must be used after {script}, like for the :append and :insert commands. Refer to :let-heredoc for more information.

Example:

```
function! IcecreamInitialize()
python << EOF
class StrawberryIcecream:
        def __call__(self):
                print('EAT ME')
EOF
endfunction
```

To see what version of Python you have:

```
:python print(sys.version)
```

There is no need to "import sys", it's done by default.

python-environment

Environment variables set in Vim are not always available in Python. This depends on how Vim and Python were build. Also see https://docs.python.org/3/library/os.html#os.environ

Note: Python is very sensitive to indenting. Make sure the "class" line and "EOF" do not have any indent.

:pydo

:[range]pydo {body}
Execute Python function "def _vim_pydo(line, linenr): {body}" for each line in the [range], with the function arguments being set to the text of each line in turn, without a trailing <EOL>, and the current line number. The function should return a string or None. If a string is returned, it becomes the text of the line in the current turn. The default for [range] is the whole file: "1,$".

Examples:

```
:pydo return "%s\t%d" % (line[::-1], len(line))
:pydo if line: return "%4d: %s" % (linenr, line)
```

One can use :pydo in possible conjunction with :py to filter a range using python. For example:

```

:py3 << EOF
needle = vim.eval('@a')
replacement = vim.eval('@b')

def py_vim_string_replace(str):
        return str.replace(needle, replacement)
EOF
:'<,'>py3do return py_vim_string_replace(line)
```

:pyfile :pyf

:[range]pyf[ile] {file}
Execute the Python script in {file}. The whole argument is used as a single file name.

Both of these commands do essentially the same thing - they execute a piece of Python code, with the "current range" python-range set to the given line range.

In the case of :python, the code to execute is in the command-line.
In the case of :pyfile, the code to execute is the contents of the given file.

Python commands cannot be used in the sandbox.

To pass arguments you need to set sys.argv[] explicitly. Example:

```
:python sys.argv = ["foo", "bar"]
:pyfile myscript.py
```

Here are some examples python-examples

```
:python from vim import *
:python current.line = str.upper(current.line)
:python print("Hello")
:python str = current.buffer[42]
```

Note that changes (such as the "import" statements) persist from one command
to the next, just like the Python REPL.

script-here

When using a script language in-line, you might want to skip this when the
language isn't supported.

```
if has('python')
  python << EOF
    print("python works")
EOF
endif
```

Note that "EOF" must be at the start of the line without preceding white
space.

The vim module python-vim

Python code gets all of its access to vim (with one exception - see
python-output below) via the "vim" module. The vim module implements two
methods, three constants, and one error object. You need to import the vim
module before using it:

```
:python import vim
```

Overview

```
:py print("Hello")              # displays a message
:py vim.command(cmd)            # execute an Ex command
:py w = vim.windows[n]          # gets window "n"
:py cw = vim.current.window     # gets the current window
:py b = vim.buffers[n]          # gets buffer "n"
:py cb = vim.current.buffer     # gets the current buffer
:py w.height = lines            # sets the window height
:py w.cursor = (row, col)       # sets the window cursor position
:py pos = w.cursor              # gets a tuple (row, col)
:py name = b.name               # gets the buffer file name
:py line = b[n]                 # gets a line from the buffer
:py lines = b[n:m]              # gets a list of lines
:py num = len(b)                # gets the number of lines
:py b[n] = str                  # sets a line in the buffer
:py b[n:m] = [str1, str2, str3] # sets a number of lines at once
:py del b[n]                    # deletes a line
:py del b[n:m]                  # deletes a number of lines
```

Methods of the "vim" module

vim.command(str) python-command

        Executes the Ex command str. Returns None.
        Examples:

```
:py vim.command("set tw=72")
:py vim.command("%s/aaa/bbb/g")
```

        The following definition executes Normal mode commands:

```
def normal(str):
        vim.command("normal "+str)
# Note the use of single quotes to delimit a string containing
# double quotes
normal('"a2dd"aP')
```

vim.eval(str) python-eval

        Evaluates the expression str using the vim internal expression
        evaluator (see expression). Returns the expression result as:

- a string if the Vim expression evaluates to a string or number

- a list if the Vim expression evaluates to a Vim list
- a dictionary if the Vim expression evaluates to a Vim dict
- a boolean if Vim expression evaluates to v:true or v:false
- None if Vim expression evaluates to v:null

Dictionaries and lists are recursively expanded.
Examples:

```
:py text_width = vim.eval("&tw")
:py str = vim.eval("12+12")             # NB result is a string! Use
                                  # int() to convert to a
                                  # number.
```

vim.strwidth(str) python-strwidth

Like strwidth(): returns number of display cells str occupies, tab is counted as one cell.

vim.foreach_rtp(callable) python-foreach_rtp

Call the given callable for each path in 'runtimepath' until either callable returns something but None, the exception is raised or there are no longer paths. If stopped in case callable returned non-None, vim.foreach_rtp function returns the value returned by callable.

vim.chdir(*args, **kwargs) python-chdir

vim.fchdir(*args, **kwargs) python-fchdir

Run os.chdir or os.fchdir, then all appropriate vim stuff.
Note: you should not use these functions directly, use os.chdir and os.fchdir instead. Behavior of vim.fchdir is undefined in case os.fchdir does not exist.

Error object of the "vim" module

vim.error python-error

Upon encountering a Vim error, Python raises an exception of type vim.error.
Example:

```
try:
        vim.command("put a")
except vim.error:
        # nothing in register a
```

Constants of the "vim" module

Note that these are not actually constants - you could reassign them. But this is silly, as you would then lose access to the vim objects to which the variables referred.

vim.buffers python-buffers

A mapping object providing access to the list of vim buffers. The object supports the following operations:

```
:py b = vim.buffers[i]  # Indexing (read-only)
:py b in vim.buffers    # Membership test
:py n = len(vim.buffers)        # Number of elements
:py for b in vim.buffers:       # Iterating over buffer list
```

vim.windows python-windows

A sequence object providing access to the list of vim windows. The object supports the following operations:

```
:py w = vim.windows[i]  # Indexing (read-only)
:py w in vim.windows    # Membership test
:py n = len(vim.windows)        # Number of elements
:py for w in vim.windows:       # Sequential access
```

Note: vim.windows object always accesses current tab page.
python-tabpage.windows objects are bound to parent python-tabpage object and always use windows from that tab page (or throw vim.error in case tab page was deleted). You can keep a reference to both without keeping a reference to vim module object or python-tabpage, they will not lose their properties in this case.

vim.tabpages python-tabpages

A sequence object providing access to the list of vim tab pages. The object supports the following operations:

```
:py t = vim.tabpages[i] # Indexing (read-only)
:py t in vim.tabpages   # Membership test
:py n = len(vim.tabpages)       # Number of elements
:py for t in vim.tabpages:      # Sequential access
```

vim.current python-current

An object providing access (via specific attributes) to various "current" objects available in vim:

vim.current.line	The current line (RW)	String
vim.current.buffer	The current buffer (RW)	Buffer
vim.current.window	The current window (RW)	Window
vim.current.tabpage	The current tab page (RW)	TabPage
vim.current.range	The current line range (RO)	Range

The last case deserves a little explanation. When the :python or :pyfile command specifies a range, this range of lines becomes the "current range". A range is a bit like a buffer, but with all access restricted to a subset of lines. See python-range for more details.

Note: When assigning to vim.current.{buffer,window,tabpage} it expects valid python-buffer, python-window or python-tabpage objects respectively. Assigning triggers normal (with autocommands) switching to given buffer, window or tab page. It is the only way to switch UI objects in python: you can't assign to python-tabpage.window attribute. To switch without triggering autocommands use

```
py << EOF
saved_eventignore = vim.options['eventignore']
vim.options['eventignore'] = 'all'
try:
    vim.current.buffer = vim.buffers[2] # Switch to buffer 2
finally:
    vim.options['eventignore'] = saved_eventignore
EOF
```

vim.vars python-vars

vim.vvars python-vvars

Dictionary-like objects holding dictionaries with global (g:) and vim (v:) variables respectively.

vim.options python-options

Object partly supporting mapping protocol (supports setting and getting items) providing a read-write access to global options. **Note:** unlike :set this provides access only to global options. You cannot use this object to obtain or set local options' values or access local-only options in any fashion. Raises KeyError if no global option with such name exists (i.e. does not raise KeyError for global-local options and global only options, but does for window- and buffer-local ones). Use python-buffer objects to access to buffer-local options and python-window objects to access to window-local options.

Type of this object is available via "Options" attribute of vim module.

Output from Python python-output

Vim displays all Python code output in the Vim message area. Normal output appears as information messages, and error output appears as error messages.

In implementation terms, this means that all output to sys.stdout (including the output from print statements) appears as information messages, and all output to sys.stderr (including error tracebacks) appears as error messages.

python-input

Input (via sys.stdin, including input() and raw_input()) is not supported, and may cause the program to crash. This should probably be fixed.

python3-directory pythonx-directory

Python 'runtimepath' handling python-special-path

In python vim.VIM_SPECIAL_PATH special directory is used as a replacement for the list of paths found in 'runtimepath': with this directory in sys.path and vim.path_hooks in sys.path_hooks python will try to load module from {rtp}/python3 and {rtp}/pythonx for each {rtp} found in 'runtimepath'.

Implementation is similar to the following:

```
from imp import find_module, load_module
import vim
import sys

class VimModuleLoader(object):
    def __init__(self, module):
        self.module = module

    def load_module(self, fullname, path=None):
        return self.module

def _find_module(fullname, oldtail, path):
    idx = oldtail.find('.')
    if idx > 0:
        name = oldtail[:idx]
        tail = oldtail[idx+1:]
        fmr = find_module(name, path)
        module = load_module(fullname[:-len(oldtail)] + name, *fmr)
        return _find_module(fullname, tail, module.__path__)
    else:
        fmr = find_module(fullname, path)
        return load_module(fullname, *fmr)

# It uses vim module itself in place of VimPathFinder class: it does not
# matter for python which object has find_module function attached to as
# an attribute.
class VimPathFinder(object):
    @classmethod
    def find_module(cls, fullname, path=None):
        try:
            return VimModuleLoader(_find_module(fullname, fullname, path or vim._get_paths()))
        except ImportError:
            return None

    @classmethod
    def load_module(cls, fullname, path=None):
        return _find_module(fullname, fullname, path or vim._get_paths())

def hook(path):
    if path == vim.VIM_SPECIAL_PATH:
        return VimPathFinder
    else:
        raise ImportError

sys.path_hooks.append(hook)
```

vim.VIM_SPECIAL_PATH python-VIM_SPECIAL_PATH

String constant used in conjunction with vim path hook. If path hook installed by vim is requested to handle anything but path equal to vim.VIM_SPECIAL_PATH constant it raises ImportError. In the only other case it uses special loader.

Note: you must not use value of this constant directly, always use vim.VIM_SPECIAL_PATH object.

vim.find_module(...) python-find_module

vim.path_hook(path) python-path_hook

Methods or objects used to implement path loading as described above. You should not be using any of these directly except for vim.path_hook in case you need to do something with sys.meta_path. It is not guaranteed that any of the objects will exist in the future vim versions.

vim._get_paths python-_get_paths

Methods returning a list of paths which will be searched for by path hook. You should not rely on this method being present in future versions, but can use it for debugging.

It returns a list of {rtp}/python3 and {rtp}/pythonx directories for each {rtp} in 'runtimepath'.

Buffer objects python-buffer

Buffer objects represent vim buffers. You can obtain them in a number of ways:

- via vim.current.buffer (python-current)
- from indexing vim.buffers (python-buffers)
- from the "buffer" attribute of a window (python-window)

Buffer objects have two read-only attributes - name - the full file name for the buffer, and number - the buffer number. They also have three methods (append, mark, and range; see below).

You can also treat buffer objects as sequence objects. In this context, they act as if they were lists (yes, they are mutable) of strings, with each element being a line of the buffer. All of the usual sequence operations, including indexing, index assignment, slicing and slice assignment, work as you would expect. Note that the result of indexing (slicing) a buffer is a string (list of strings). This has one unusual consequence - b[:] is different from b. In particular, "b[:] = None" deletes the whole of the buffer, whereas "b = None" merely updates the variable b, with no effect on the buffer.

Buffer indexes start at zero, as is normal in Python. This differs from vim line numbers, which start from 1. This is particularly relevant when dealing with marks (see below) which use vim line numbers.

The buffer object attributes are:

b.vars	Dictionary-like object used to access buffer-variables.
b.options	Mapping object (supports item getting, setting and deleting) that provides access to buffer-local options and buffer-local values of global-local options. Use python-window.options if option is window-local, this object will raise KeyError. If option is global-local and local value is missing getting it will return None.
b.name	String, RW. Contains buffer name (full path). **Note:** when assigning to b.name BufFilePre and BufFilePost autocommands are launched.
b.number	Buffer number. Can be used as python-buffers key. Read-only.
b.valid	True or False. Buffer object becomes invalid when corresponding buffer is wiped out.

The buffer object methods are:

b.append(str)	Append a line to the buffer
b.append(str, nr)	Idem, below line "nr"
b.append(list)	Append a list of lines to the buffer Note that the option of supplying a list of strings to the append method differs from the equivalent method for Python's built-in list objects.
b.append(list, nr)	Idem, below line "nr"
b.mark(name)	Return a tuple (row,col) representing the position of the named mark (can also get the []"<> marks)
b.range(s,e)	Return a range object (see python-range) which represents the part of the given buffer between line numbers s and e inclusive.

Note that when adding a line it must not contain a line break character '\n'. A trailing '\n' is allowed and ignored, so that you can do:

```
:py b.append(f.readlines())
```

Buffer object type is available using "Buffer" attribute of vim module.

Examples (assume b is the current buffer)

```
:py print(b.name)               # write the buffer file name
:py b[0] = "hello!!!"           # replace the top line
:py b[:] = None                 # delete the whole buffer
:py del b[:]                    # delete the whole buffer
:py b[0:0] = [ "a line" ]       # add a line at the top
:py del b[2]                    # delete a line (the third)
:py b.append("bottom")          # add a line at the bottom
:py n = len(b)                  # number of lines
:py (row,col) = b.mark('a')     # named mark
:py r = b.range(1,5)            # a sub-range of the buffer
:py b.vars["foo"] = "bar"       # assign b:foo variable
:py b.options["ff"] = "dos"     # set fileformat
:py del b.options["ar"]         # same as :set autoread<
```

Range objects

python-range

Range objects represent a part of a vim buffer. You can obtain them in a number of ways:

- via vim.current.range (python-current)
- from a buffer's range() method (python-buffer)

A range object is almost identical in operation to a buffer object. However, all operations are restricted to the lines within the range (this line range can, of course, change as a result of slice assignments, line deletions, or the range.append() method).

The range object attributes are:

r.start	Index of first line into the buffer

```
        r.end           Index of last line into the buffer
```

The range object methods are:

```
        r.append(str)   Append a line to the range
        r.append(str, nr)  Idem, after line "nr"
        r.append(list)  Append a list of lines to the range
                        Note that the option of supplying a list of strings to
                        the append method differs from the equivalent method
                        for Python's built-in list objects.
        r.append(list, nr)  Idem, after line "nr"
```

Range object type is available using "Range" attribute of vim module.

Example (assume r is the current range):

```
        # Send all lines in a range to the default printer
        vim.command("%d,%dhardcopy!" % (r.start+1,r.end+1))
```

Window objects

python-window

Window objects represent vim windows. You can obtain them in a number of ways:

- via vim.current.window (python-current)
- from indexing vim.windows (python-windows)
- from indexing "windows" attribute of a tab page (python-tabpage)
- from the "window" attribute of a tab page (python-tabpage)

You can manipulate window objects only through their attributes. They have no methods, and no sequence or other interface.

Window attributes are:

```
        buffer (read-only)      The buffer displayed in this window
        cursor (read-write)     The current cursor position in the window
                                This is a tuple, (row,col).
        height (read-write)     The window height, in rows
        width (read-write)      The window width, in columns
        vars (read-only)        The window w: variables. Attribute is
                                unassignable, but you can change window
                                variables this way
        options (read-only)     The window-local options. Attribute is
                                unassignable, but you can change window
                                options this way. Provides access only to
                                window-local options, for buffer-local use
                                python-buffer and for global ones use
                                python-options. If option is global-local
                                and local value is missing getting it will
                                return None.
        number (read-only)      Window number.  The first window has number 1.
                                This is zero in case it cannot be determined
                                (e.g. when the window object belongs to other
                                tab page).
        row, col (read-only)    On-screen window position in display cells.
                                First position is zero.
        tabpage (read-only)     Window tab page.
        valid (read-write)      True or False. Window object becomes invalid
                                when corresponding window is closed.
```

The height attribute is writable only if the screen is split horizontally.
The width attribute is writable only if the screen is split vertically.

Window object type is available using "Window" attribute of vim module.

Tab page objects

python-tabpage

Tab page objects represent vim tab pages. You can obtain them in a number of ways:

- via vim.current.tabpage (python-current)
- from indexing vim.tabpages (python-tabpages)

You can use this object to access tab page windows. They have no methods and no sequence or other interfaces.

Tab page attributes are:

```
        number          The tab page number like the one returned by
                        tabpagenr().
        windows         Like python-windows, but for current tab page.
        vars            The tab page t: variables.
        window          Current tabpage window.
        valid           True or False. Tab page object becomes invalid when
                        corresponding tab page is closed.
```

TabPage object type is available using "TabPage" attribute of vim module.

pyeval() and py3eval() Vim functions

python-pyeval

To facilitate bi-directional interface, you can use pyeval() and py3eval()
functions to evaluate Python expressions and pass their values to Vim script.
pyxeval() is also available.

Python 3

python3

As Python 3 is the only supported version in Nvim, "python" is synonymous
with "python3" in the current version. However, code that aims to support older
versions of Nvim, as well as Vim, should prefer to use "python3" variants
explicitly if Python 3 is required.

:py3 :python3

```
:[range]py3 {stmt}
:[range]py3 << [trim] [{endmarker}]
{script}
{endmarker}
```

```
:[range]python3 {stmt}
:[range]python3 << [trim] [{endmarker}]
{script}
{endmarker}
```

The :py3 and :python3 commands work similar to :python. A
simple check if the :py3 command is working:

```
:py3 print("Hello")
```

To see what version of Python you have:

```
:py3 import sys
:py3 print(sys.version)
```

:py3file

```
:[range]py3f[ile] {file}
```

The :py3file command works similar to :pyfile.

:py3do

```
:[range]py3do {body}
```

The :py3do command works similar to :pydo.

E880

Raising SystemExit exception in python isn't endorsed way to quit vim, use:

```
:py vim.command("qall!")
```

has-python

You can test if Python is available with:

```
if has('pythonx')
  echo 'there is Python'
endif
if has('python3')
  echo 'there is Python 3.x'
endif
```

Python 2 is no longer supported. Thus has('python') always returns
zero for backwards compatibility reasons.

Python X

python_x pythonx

The "pythonx" and "pyx" prefixes were introduced for python code which
works with Python 2.6+ and Python 3. As Nvim only supports Python 3,
all these commands are now synonymous to their "python3" equivalents.

:pyx :pythonx

:pyx and :pythonx work the same as :python3. To check if :pyx works:

```
:pyx print("Hello")
```

To see what version of Python is being used:

```
:pyx import sys
:pyx print(sys.version)
```

:pyxfile python_x-special-comments

:pyxfile works the same as :py3file.

:pyxdo

:pyxdo works the same as :py3do.

has-pythonx

To check if pyx* functions and commands are available:

```
if has('pythonx')
  echo 'pyx* commands are available. (Python ' .. &pyx .. ')'
endif
```

The Ruby Interface to Vim ruby Ruby

See provider-ruby for more information.

E266 E267 E268 E269 E270 E271 E272 E273

The home page for ruby is https://www.ruby-lang.org/. You can find links for downloading Ruby there.

1. Commands ruby-commands

:ruby :rub

:rub[y] {cmd} Execute Ruby command {cmd}. A command to try it out:

```
:ruby print "Hello"
```

:rub[y] << [trim] [{endmarker}]
{script}
{endmarker}

Execute Ruby script {script}.

If [endmarker] is omitted, it defaults to a dot '.' like for the :append and :insert commands. Refer to :let-heredoc for more information.

This form of the :ruby command is mainly useful for including ruby code in vim scripts.

Example Vim script:

```
function! RedGem()
ruby << EOF
class Garnet
        def initialize(s)
                @buffer = VIM::Buffer.current
                vimputs(s)
        end
        def vimputs(s)
                @buffer.append(@buffer.count,s)
        end
end
gem = Garnet.new("pretty")
EOF
endfunction
```

To see what version of Ruby you have:

```
:ruby print RUBY_VERSION
```

:rubydo :rubyd E265

:[range]rubyd[o] {cmd} Evaluate Ruby command {cmd} for each line in the [range], with $_ being set to the text of each line in turn, without a trailing <EOL>. Setting $_ will change the text, but note that it is not possible to add or delete lines using this command.
The default for [range] is the whole file: "1,$".

:rubyfile :rubyf

:rubyf[ile] {file} Execute the Ruby script in {file}. This is the same as :ruby load 'file', but allows file name completion.

Executing Ruby commands is not possible in the sandbox.

2. The VIM module ruby-vim

Ruby code gets all of its access to vim via the "VIM" module.

Overview

```
print "Hello"                          # displays a message
VIM.command(cmd)                       # execute an Ex command
num = VIM::Window.count                # gets the number of windows
w = VIM::Window[n]                     # gets window "n"
cw = VIM::Window.current               # gets the current window
num = VIM::Buffer.count                # gets the number of buffers
b = VIM::Buffer[n]                     # gets buffer "n"
cb = VIM::Buffer.current               # gets the current buffer
w.height = lines                       # sets the window height
w.cursor = [row, col]                  # sets the window cursor position
pos = w.cursor                         # gets an array [row, col]
name = b.name                          # gets the buffer file name
line = b[n]                            # gets a line from the buffer
num = b.count                          # gets the number of lines
b[n] = str                             # sets a line in the buffer
b.delete(n)                            # deletes a line
b.append(n, str)                       # appends a line after n
line = VIM::Buffer.current.line        # gets the current line
num = VIM::Buffer.current.line_number # gets the current line number
VIM::Buffer.current.line = "test"      # sets the current line number
```

Module Functions:

ruby-message

VIM::message({msg})
        Displays the message {msg}.

ruby-set_option

VIM::set_option({arg})
        Sets a vim option. {arg} can be any argument that the ":set" command
        accepts. Note that this means that no spaces are allowed in the
        argument! See :set.

ruby-command

VIM::command({cmd})
        Executes Ex command {cmd}.

ruby-evaluate

VIM::evaluate({expr})
        Evaluates {expr} using the vim internal expression evaluator (see
        expression). Returns the expression result as a string.
        A List is turned into a string by joining the items and inserting
        line breaks.

3. VIM::Buffer objects ruby-buffer

VIM::Buffer objects represent vim buffers.

Class Methods:

```
current         Returns the current buffer object.
count           Returns the number of buffers.
self[{n}]       Returns the buffer object for the number {n}.  The first
                number is 0.
```

Methods:

```
name            Returns the full name of the buffer.
number          Returns the number of the buffer.
count           Returns the number of lines.
length          Returns the number of lines.
self[{n}]       Returns a line from the buffer.  {n} is the line number.
self[{n}] = {str}
                Sets a line in the buffer.  {n} is the line number.
delete({n})     Deletes a line from the buffer.  {n} is the line number.
append({n}, {str})
                Appends a line after the line {n}.
line            Returns the current line of the buffer if the buffer is
                active.
line = {str}    Sets the current line of the buffer if the buffer is active.
line_number     Returns the number of the current line if the buffer is
                active.
```

4. VIM::Window objects ruby-window

VIM::Window objects represent vim windows.

Class Methods:

```
current         Returns the current window object.
count           Returns the number of windows.
self[{n}]       Returns the window object for the number {n}.  The first
                number is 0.
```

Methods:

```
buffer          Returns the buffer displayed in the window.
height          Returns the height of the window.
height = {n}    Sets the window height to {n}.
width           Returns the width of the window.
width = {n}     Sets the window width to {n}.
cursor          Returns a [row, col] array for the cursor position.
                First line number is 1 and first column number is 0.
cursor = [{row}, {col}]
                Sets the cursor position to {row} and {col}.
```

5. Global variables ruby-globals

There are two global variables.

```
$curwin         The current window object.
$curbuf         The current buffer object.
```

6. rubyeval() Vim function ruby-rubyeval

To facilitate bi-directional interface, you can use rubyeval() function to
evaluate Ruby expressions and pass their values to Vim script.

The Ruby value "true", "false" and "nil" are converted to v:true, v:false and
v:null, respectively.

This file contains a list of all commands for each mode, with a tag and a
short description. The lists are sorted on ASCII value.

Tip: When looking for certain functionality, use a search command. E.g.,
to look for deleting something, use: "/delete".

For an overview of options see option-list.
For an overview of built-in functions see functions.
For a list of Vim variables see vvars.

1. Insert mode

insert-index

Tag	Char	Insert-mode action
i_CTRL-@	CTRL-@	insert previously inserted text and stop insert
i_CTRL-A	CTRL-A	insert previously inserted text
i_CTRL-C	CTRL-C	quit insert mode, without checking for abbreviation
i_CTRL-D	CTRL-D	delete one shiftwidth of indent in the current line
i_CTRL-E	CTRL-E	insert the character which is below the cursor
	CTRL-F	not used (but by default it's in 'cinkeys' to re-indent the current line)
i_CTRL-G_j	CTRL-G CTRL-J	line down, to column where inserting started
i_CTRL-G_j	CTRL-G j	line down, to column where inserting started
i_CTRL-G_j	CTRL-G <Down>	line down, to column where inserting started
i_CTRL-G_k	CTRL-G CTRL-K	line up, to column where inserting started
i_CTRL-G_k	CTRL-G k	line up, to column where inserting started
i_CTRL-G_k	CTRL-G <Up>	line up, to column where inserting started
i_CTRL-G_u	CTRL-G u	start new undoable edit
i_CTRL-G_U	CTRL-G U	don't break undo with next cursor movement
i_<BS>	<BS>	delete character before the cursor
i_digraph	{char1}<BS>{char2} enter digraph (only when 'digraph' option set)	
i_CTRL-H	CTRL-H	same as <BS>
i_<Tab>	<Tab>	insert a <Tab> character
i_CTRL-I	CTRL-I	same as <Tab>
i_<NL>	<NL>	same as <CR>
i_CTRL-J	CTRL-J	same as <CR>
i_CTRL-K	CTRL-K {char1} {char2} enter digraph	
i_<CR>	<CR>	begin new line
i_CTRL-M	CTRL-M	same as <CR>
i_CTRL-N	CTRL-N	find next match for keyword in front of the cursor
i_CTRL-O	CTRL-O	execute a single command and return to insert mode
i_CTRL-P	CTRL-P	find previous match for keyword in front of the cursor
i_CTRL-Q	CTRL-Q	same as CTRL-V, unless used for terminal control flowi_CTRL-SHIFT-Q CTRL-SHIFT-Q {char} like CTRL-Q unless tui-modifyOtherKeys is active
i_CTRL-R	CTRL-R {register} insert the contents of a registeri_CTRL-R_CTRL-R CTRL-R CTRL-R {register} insert the contents of a register literallyi_CTRL-R_CTRL-O CTRL-R CTRL-O {register} insert the contents of a register literally and don't auto-indenti_CTRL-R_CTRL-P CTRL-R CTRL-P {register} insert the contents of a register literally and fix indent.	
	CTRL-S	not used or used for terminal control flow
i_CTRL-T	CTRL-T	insert one shiftwidth of indent in current line
i_CTRL-U	CTRL-U	delete all entered characters in the current line
i_CTRL-V	CTRL-V {char}	insert next non-digit literallyi_CTRL-SHIFT-V CTRL-SHIFT-V {char} like CTRL-V unless tui-modifyOtherKeys is activei_CTRL-V_digit CTRL-V {number} insert three digit decimal number as a single byte.

Tag	Char	Insert-mode action
i_CTRL-W	CTRL-W	delete word before the cursor
i_CTRL-X	CTRL-X {mode}	enter CTRL-X sub mode, see i_CTRL-X_index
i_CTRL-Y	CTRL-Y	insert the character which is above the cursor
i_<Esc>	<Esc>	end insert mode
i_CTRL-[	CTRL-[	same as <Esc>
i_CTRL-_CTRL-N CTRL-\ CTRL-N	go to Normal mode	
i_CTRL-_CTRL-G CTRL-\ CTRL-G	go to Normal mode	
	CTRL-\ a - z	reserved for extensions
	CTRL-\ others	not used
i_CTRL-]	CTRL-]	trigger abbreviation
i_CTRL-^	CTRL-^	toggle use of :lmap mappings
i_CTRL-_	CTRL-_	When 'allowrevins' set: toggle 'revins'
	<Space> to '~'	not used, except '0' and '^' followed by CTRL-D
i_0_CTRL-D	0 CTRL-D	delete all indent in the current line
i_^_CTRL-D	^ CTRL-D	delete all indent in the current line, restore it in the next line
i_<Del>	<Del>	delete character under the cursor
	Meta characters (0x80 to 0xff, 128 to 255)	not used
i_<Left>	<Left>	cursor one character left
i_<S-Left>	<S-Left>	cursor one word left
i_<C-Left>	<C-Left>	cursor one word left
i_<Right>	<Right>	cursor one character right
i_<S-Right>	<S-Right>	cursor one word right
i_<C-Right>	<C-Right>	cursor one word right
i_<Up>	<Up>	cursor one line up
i_<S-Up>	<S-Up>	same as <PageUp>
i_<Down>	<Down>	cursor one line down
i_<S-Down>	<S-Down>	same as <PageDown>
i_<Home>	<Home>	cursor to start of line
i_<C-Home>	<C-Home>	cursor to start of file
i_<End>	<End>	cursor past end of line
i_<C-End>	<C-End>	cursor past end of file
i_<PageUp>	<PageUp>	one screenful backward
i_<PageDown>	<PageDown>	one screenful forward
i_<F1>	<F1>	same as <Help>
i_<Help>	<Help>	stop insert mode and display help window
i_<Insert>	<Insert>	toggle Insert/Replace mode
i_<LeftMouse>	<LeftMouse>	cursor at mouse click
i_<ScrollWheelDown>	<ScrollWheelDown>	move window three lines down
i_<S-ScrollWheelDown>	<S-ScrollWheelDown>	move window one page down
i_<ScrollWheelUp>	<ScrollWheelUp>	move window three lines up
i_<S-ScrollWheelUp>	<S-ScrollWheelUp>	move window one page up
i_<ScrollWheelLeft>	<ScrollWheelLeft>	move window six columns left
i_<S-ScrollWheelLeft>	<S-ScrollWheelLeft>	move window one page left
i_<ScrollWheelRight>	<ScrollWheelRight>	move window six columns right
i_<S-ScrollWheelRight> <S-ScrollWheelRight>	move window one page right	

Tag	Char	CTRL-X submode action
i_CTRL-X_CTRL-D	CTRL-X CTRL-D	complete defined identifiers
i_CTRL-X_CTRL-E	CTRL-X CTRL-E	scroll up
i_CTRL-X_CTRL-F	CTRL-X CTRL-F	complete file names
i_CTRL-X_CTRL-I	CTRL-X CTRL-I	complete identifiers
i_CTRL-X_CTRL-K	CTRL-X CTRL-K	complete identifiers from dictionary
i_CTRL-X_CTRL-L	CTRL-X CTRL-L	complete whole lines
i_CTRL-X_CTRL-N	CTRL-X CTRL-N	next completion
i_CTRL-X_CTRL-O	CTRL-X CTRL-O	omni completion
i_CTRL-X_CTRL-P	CTRL-X CTRL-P	previous completion
i_CTRL-X_CTRL-R	CTRL-X CTRL-R	complete contents from registers
i_CTRL-X_CTRL-S	CTRL-X CTRL-S	spelling suggestions
i_CTRL-X_CTRL-T	CTRL-X CTRL-T	complete identifiers from thesaurus
i_CTRL-X_CTRL-Y	CTRL-X CTRL-Y	scroll down
i_CTRL-X_CTRL-U	CTRL-X CTRL-U	complete with 'completefunc'
i_CTRL-X_CTRL-V	CTRL-X CTRL-V	complete like in : command line
i_CTRL-X_CTRL-Z	CTRL-X CTRL-Z	stop completion, text is unchanged
i_CTRL-X_CTRL-]	CTRL-X CTRL-]	complete tags
i_CTRL-X_s	CTRL-X s	spelling suggestions

commands in completion mode (see popupmenu-keys)

Tag	Char	Completion-mode action
complete_CTRL-E	CTRL-E	stop completion and go back to original text
complete_CTRL-Y	CTRL-Y	accept selected match and stop completion
	CTRL-L	insert one character from the current match
	<CR>	insert currently selected match
	<BS>	delete one character and redo search
	CTRL-H	same as <BS>
	<Up>	select the previous match
	<Down>	select the next match
	<PageUp>	select a match several entries back
	<PageDown>	select a match several entries forward
	other	stop completion and insert the typed character

2. Normal mode

normal-index

```
CHAR     any non-blank character
WORD     a sequence of non-blank characters
N        a number entered before the command
{motion} a cursor movement command
Nmove    the text that is moved over with a {motion}
SECTION  a section that possibly starts with '}' instead of '{'
```

note: 1 = cursor movement command; 2 = can be undone/redone

Tag	Char	Note Normal-mode action
	CTRL-@	not used
CTRL-A	CTRL-A	2 add N to number at/after cursor
CTRL-B	CTRL-B	1 scroll N screens Backwards
CTRL-C	CTRL-C	interrupt current (search) command
CTRL-D	CTRL-D	scroll Down N lines (default: half a screen)
CTRL-E	CTRL-E	scroll N lines upwards (N lines Extra)
CTRL-F	CTRL-F	1 scroll N screens Forward
CTRL-G	CTRL-G	display current file name and position
<BS>	<BS>	1 same as "h"
CTRL-H	CTRL-H	1 same as "h"
<Tab>	<Tab>	1 go to N newer entry in jump list
CTRL-I	CTRL-I	1 same as <Tab>
<NL>	<NL>	1 same as "j"
<S-NL>	<S-NL>	1 same as CTRL-F
CTRL-J	CTRL-J	1 same as "j"
	CTRL-K	not used
CTRL-L	CTRL-L	redraw screen
<CR>	<CR>	1 cursor to the first CHAR N lines lower
<S-CR>	<S-CR>	1 same as CTRL-F
CTRL-M	CTRL-M	1 same as <CR>
CTRL-N	CTRL-N	1 same as "j"
CTRL-O	CTRL-O	1 go to N older entry in jump list
CTRL-P	CTRL-P	1 same as "k"
	CTRL-Q	not used, or used for terminal control flow
CTRL-R	CTRL-R	2 redo changes which were undone with 'u'
	CTRL-S	not used, or used for terminal control flow
CTRL-T	CTRL-T	jump to N older Tag in tag list
CTRL-U	CTRL-U	scroll N lines Upwards (default: half a screen)
CTRL-V	CTRL-V	start blockwise Visual mode
CTRL-W	CTRL-W {char}	window commands, see CTRL-W
CTRL-X	CTRL-X	2 subtract N from number at/after cursor
CTRL-Y	CTRL-Y	scroll N lines downwards
CTRL-Z	CTRL-Z	suspend program (or start new shell)
	CTRL-[<Esc>	not used
CTRL-_CTRL-N	CTRL-\ CTRL-N	go to Normal mode (no-op)
CTRL-_CTRL-G	CTRL-\ CTRL-G	go to Normal mode (no-op)
	CTRL-\ a - z	reserved for extensions
	CTRL-\ others	not used
CTRL-]	CTRL-]	:ta to ident under cursor
CTRL-^	CTRL-^	edit Nth alternate file (equivalent to ":e #N")
CTRL-<Tab>	CTRL-<Tab>	same as g<Tab> : go to last accessed tab page
	CTRL-_	not used
<Space>	<Space>	1 same as "l"
!	!{motion}{filter} 2 filter Nmove text through the {filter} command	
!!	!!{filter}	2 filter N lines through the {filter} command
quote	"{register}	use {register} for next delete, yank or put ({.%#:} only work with put)
#	#	1 search backward for the Nth occurrence of the ident under the cursor
$	$	1 cursor to the end of Nth next line
%	%	1 find the next (curly/square) bracket on this line and go to its match, or go to matching comment bracket, or go to matching preprocessor directive.
N%	{count}%	1 go to N percentage in the file
&	&	2 repeat last :s
'	'{a-zA-Z0-9}	1 cursor to the first CHAR on the line with mark {a-zA-Z0-9}
''	''	1 cursor to the first CHAR of the line where the cursor was before the latest jump.
'(	'(	1 cursor to the first CHAR on the line of the start of the current sentence

Tag	Char	Note Normal-mode action
')	')	1 cursor to the first CHAR on the line of the end of the current sentence
'<	'<	1 cursor to the first CHAR of the line where highlighted area starts/started in the current buffer.
'>	'>	1 cursor to the first CHAR of the line where highlighted area ends/ended in the current buffer.
'[	'[	1 cursor to the first CHAR on the line of the start of last operated text or start of put text
']	']	1 cursor to the first CHAR on the line of the end of last operated text or end of put text
'{	'{	1 cursor to the first CHAR on the line of the start of the current paragraph
'}	'}	1 cursor to the first CHAR on the line of the end of the current paragraph
(	(	1 cursor N sentences backward
)	)	1 cursor N sentences forward
star	*	1 search forward for the Nth occurrence of the ident under the cursor
+	+	1 same as <CR>
<S-Plus>	<S-+>	1 same as CTRL-F
,	,	1 repeat latest f, t, F or T in opposite direction N times
-	-	1 cursor to the first CHAR N lines higher
<S-Minus>	<S-->	1 same as CTRL-B
.	.	2 repeat last change with count replaced with N
/	/{pattern}<CR>	1 search forward for the Nth occurrence of {pattern}
/<CR>	/<CR>	1 search forward for {pattern} of last search
0	0	1 cursor to the first char of the line
count	1	prepend to command to give a count
count	2	"
count	3	"
count	4	"
count	5	"
count	6	"
count	7	"
count	8	"
count	9	"
:	:	1 start entering an Ex command
N:	{count}:	start entering an Ex command with range from current line to N-1 lines down
;	;	1 repeat latest f, t, F or T N times
<	<{motion}	2 shift Nmove lines one 'shiftwidth' leftwards
<<	<<	2 shift N lines one 'shiftwidth' leftwards
=	={motion}	2 filter Nmove lines through "indent"
==	==	2 filter N lines through "indent"
>	>{motion}	2 shift Nmove lines one 'shiftwidth' rightwards
>>	>>	2 shift N lines one 'shiftwidth' rightwards
?	?{pattern}<CR>	1 search backward for the Nth previous occurrence of {pattern}
?<CR>	?<CR>	1 search backward for {pattern} of last search
@	@{a-z}	2 execute the contents of register {a-z} N times
@:	@:	repeat the previous ":" command N times
@@	@@	2 repeat the previous @{a-z} N times
A	A	2 append text after the end of the line N times
B	B	1 cursor N WORDS backward
C	["x]C	2 change from the cursor position to the end of the line, and N-1 more lines [into register x]; synonym for "c$"
D	["x]D	2 delete the characters under the cursor until the end of the line and N-1 more lines [into register x]; synonym for "d$"
E	E	1 cursor forward to the end of WORD N
F	F{char}	1 cursor to the Nth occurrence of {char} to the left
G	G	1 cursor to line N, default last line
H	H	1 cursor to line N from top of screen
I	I	2 insert text before the first CHAR on the line N times

Tag	Char	Note Normal-mode action
J	J	2 Join N lines; default is 2
K	K	lookup Keyword under the cursor with 'keywordprg'
L	L	1 cursor to line N from bottom of screen
M	M	1 cursor to middle line of screen
N	N	1 repeat the latest '/' or '?' N times in opposite direction
O	O	2 begin a new line above the cursor and insert text, repeat N times
P	["x]P	2 put the text [from register x] before the cursor N times
R	R	2 enter replace mode: overtype existing characters, repeat the entered text N-1 times
S	["x]S	2 delete N lines [into register x] and start insert; synonym for "cc".
T	T{char}	1 cursor till after Nth occurrence of {char} to the left
U	U	2 undo all latest changes on one line
V	V	start linewise Visual mode
W	W	1 cursor N WORDS forward
X	["x]X	2 delete N characters before the cursor [into register x]
Y	["x]Y	yank N lines [into register x]; synonym for "yy" **Note:** Mapped to "y$" by default. default-mappings
ZZ	ZZ	write if buffer changed and close window
ZQ	ZQ	close window without writing
[	[{char}	square bracket command (see [below)
	\	not used
]	]{char}	square bracket command (see] below)
^	^	1 cursor to the first CHAR of the line
_	_	1 cursor to the first CHAR N - 1 lines lower
`	`{a-zA-Z0-9}	1 cursor to the mark {a-zA-Z0-9}
`(	`(	1 cursor to the start of the current sentence
`)	`)	1 cursor to the end of the current sentence
`<	`<	1 cursor to the start of the highlighted area
`>	`>	1 cursor to the end of the highlighted area
`[	`[	1 cursor to the start of last operated text or start of putted text
`]	`]	1 cursor to the end of last operated text or end of putted text
``	"``"	1 cursor to the position before latest jump
`{	`{	1 cursor to the start of the current paragraph
`}	`}	1 cursor to the end of the current paragraph
a	a	2 append text after the cursor N times
b	b	1 cursor N words backward
c	["x]c{motion}	2 delete Nmove text [into register x] and start insert
cc	["x]cc	2 delete N lines [into register x] and start insert
d	["x]d{motion}	2 delete Nmove text [into register x]
dd	["x]dd	2 delete N lines [into register x]
do	do	2 same as ":diffget"
dp	dp	2 same as ":diffput"
e	e	1 cursor forward to the end of word N
f	f{char}	1 cursor to Nth occurrence of {char} to the right
g	g{char}	extended commands, see g below
h	h	1 cursor N chars to the left
i	i	2 insert text before the cursor N times
j	j	1 cursor N lines downward
k	k	1 cursor N lines upward
l	l	1 cursor N chars to the right
m	m{A-Za-z}	set mark {A-Za-z} at cursor position
n	n	1 repeat the latest '/' or '?' N times

Tag	Char	Note Normal-mode action
o	o	2 begin a new line below the cursor and insert text, repeat N times
p	["x]p	2 put the text [from register x] after the cursor N times
q	q{0-9a-zA-Z"}	record typed characters into named register {0-9a-zA-Z"} (uppercase to append)
q	q	(while recording) stops recording
Q	Q	place/remove a multicursor; with [count]: place at each search match
q:	[count]q:	edit : command-line in command-line window, with [count]: switch to Ex-mode
q/	q/	edit / command-line in command-line window
q?	q?	edit ? command-line in command-line window
r	r{char}	2 replace N chars with {char}
s	["x]s	2 (substitute) delete N characters [into register x] and start insert
t	t{char}	1 cursor till before Nth occurrence of {char} to the right
u	u	2 undo changes
v	v	start charwise Visual mode
w	w	1 cursor N words forward
x	["x]x	2 delete N characters under and after the cursor [into register x]
y	["x]y{motion}	yank Nmove text [into register x]
yy	["x]yy	yank N lines [into register x]
z	z{char}	commands starting with 'z', see z below
{	{	1 cursor N paragraphs backward
bar	|	1 cursor to column N
}	}	1 cursor N paragraphs forward
~	~	2 'tildeop' off: switch case of N characters under cursor and move the cursor N characters to the right
~	~{motion}	'tildeop' on: switch case of Nmove text
<C-End>	<C-End>	1 same as "G"
<C-Home>	<C-Home>	1 same as "gg"
<C-Left>	<C-Left>	1 same as "b"
<C-LeftMouse>	<C-LeftMouse>	place a multicursor at mouse-click
<C-Right>	<C-Right>	1 same as "w"
<C-RightMouse> <C-RightMouse>	same as "CTRL-T"	
<C-Tab>	<C-Tab>	same as "g<Tab>"
<Del>	["x]<Del>	2 same as "x"
N<Del>	{count}<Del>	remove the last digit from {count}
<Down>	<Down>	1 same as "j"
<End>	<End>	1 same as "$"
<F1>	<F1>	same as <Help>
<Help>	<Help>	open a help window
<Home>	<Home>	1 same as "0"
<Insert>	<Insert>	2 same as "i"
<Left>	<Left>	1 same as "h"
<LeftMouse>	<LeftMouse>	1 move cursor to the mouse click position
<MiddleMouse>	<MiddleMouse>	2 same as "gP" at the mouse click position
<PageDown>	<PageDown>	same as CTRL-F
<PageUp>	<PageUp>	same as CTRL-B
<Right>	<Right>	1 same as "l"
<RightMouse>	<RightMouse>	start Visual mode, move cursor to the mouse click position
<S-Down>	<S-Down>	1 same as CTRL-F
<S-Left>	<S-Left>	1 same as "b"
<S-LeftMouse>	<S-LeftMouse>	same as "*" at the mouse click position
<S-Right>	<S-Right>	1 same as "w"
<S-RightMouse> <S-RightMouse>	same as "#" at the mouse click position	
<S-Up>	<S-Up>	1 same as CTRL-B
<Undo>	<Undo>	2 same as "u"
<Up>	<Up>	1 same as "k"
<ScrollWheelDown>	<ScrollWheelDown>	move window three lines down

Tag	Char	Note Normal-mode action
<S-ScrollWheelDown>	<S-ScrollWheelDown>	move window one page down
<ScrollWheelUp>	<ScrollWheelUp>	move window three lines up
<S-ScrollWheelUp>	<S-ScrollWheelUp>	move window one page up
<ScrollWheelLeft>	<ScrollWheelLeft>	move window six columns left
<S-ScrollWheelLeft>	<S-ScrollWheelLeft>	move window one page left
<ScrollWheelRight>	<ScrollWheelRight>	move window six columns right
<S-ScrollWheelRight>	<S-ScrollWheelRight>	move window one page right

2.1 Text objects

objects

These can be used after an operator or in Visual mode to select an object.

Tag	Command	Op-pending and Visual-mode action
v_aquote	a"	double quoted string
v_a'	a'	single quoted string
v_a(	a(	same as ab
v_a)	a)	same as ab
v_a<	a<	"a <>" from '<' to the matching '>'
v_a>	a>	same as a<
v_aB	aB	"a Block" from [{ to]} (with brackets)
v_aW	aW	"a WORD" (with white space)
v_a[	a[	"a []" from '[' to the matching ']'
v_a]	a]	same as a[
v_a`	a`	string in backticks
v_ab	ab	"a block" from "[(" to "])" (with braces)
v_al	al	all lines (entire buffer)
v_ap	ap	"a paragraph" (with white space)
v_as	as	"a sentence" (with white space)
v_at	at	"a tag block" (with white space)
v_aw	aw	"a word" (with white space)
v_a{	a{	same as aB
v_a}	a}	same as aB
v_iquote	i"	double quoted string without the quotes
v_i'	i'	single quoted string without the quotes
v_i(	i(	same as ib
v_i)	i)	same as ib
v_i<	i<	"inner <>" from '<' to the matching '>'
v_i>	i>	same as i<
v_iB	iB	"inner Block" from [{ and]}
v_iW	iW	"inner WORD"
v_i[	i[	"inner []" from '[' to the matching ']'
v_i]	i]	same as i[
v_i`	i`	string in backticks without the backticks
v_ib	ib	"inner block" from "[(" to "])"
v_il	il	inner line (without surrounding whitespace)
v_ip	ip	"inner paragraph"
v_is	is	"inner sentence"
v_it	it	"inner tag block"
v_iw	iw	"inner word"
v_i{	i{	same as iB
v_i}	i}	same as iB

2.2 Window commands

CTRL-W

Tag	Command	Normal-mode action
CTRL-W_CTRL-B	CTRL-W CTRL-B	same as "CTRL-W b"
CTRL-W_CTRL-C	CTRL-W CTRL-C	no-op
CTRL-W_CTRL-D	CTRL-W CTRL-D	same as "CTRL-W d"
CTRL-W_CTRL-F	CTRL-W CTRL-F	same as "CTRL-W f"
	CTRL-W CTRL-G	same as "CTRL-W g .."
CTRL-W_CTRL-H	CTRL-W CTRL-H	same as "CTRL-W h"
CTRL-W_CTRL-I	CTRL-W CTRL-I	same as "CTRL-W i"
CTRL-W_CTRL-J	CTRL-W CTRL-J	same as "CTRL-W j"
CTRL-W_CTRL-K	CTRL-W CTRL-K	same as "CTRL-W k"
CTRL-W_CTRL-L	CTRL-W CTRL-L	same as "CTRL-W l"
CTRL-W_CTRL-N	CTRL-W CTRL-N	same as "CTRL-W n"
CTRL-W_CTRL-O	CTRL-W CTRL-O	same as "CTRL-W o"
CTRL-W_CTRL-P	CTRL-W CTRL-P	same as "CTRL-W p"
CTRL-W_CTRL-Q	CTRL-W CTRL-Q	same as "CTRL-W q"
CTRL-W_CTRL-R	CTRL-W CTRL-R	same as "CTRL-W r"
CTRL-W_CTRL-S	CTRL-W CTRL-S	same as "CTRL-W s"
CTRL-W_CTRL-T	CTRL-W CTRL-T	same as "CTRL-W t"
CTRL-W_CTRL-V	CTRL-W CTRL-V	same as "CTRL-W v"
CTRL-W_CTRL-W	CTRL-W CTRL-W	same as "CTRL-W w"
CTRL-W_CTRL-X	CTRL-W CTRL-X	same as "CTRL-W x"
CTRL-W_CTRL-Z	CTRL-W CTRL-Z	same as "CTRL-W z"
CTRL-W_CTRL-]	CTRL-W CTRL-]	same as "CTRL-W]"
CTRL-W_CTRL-^	CTRL-W CTRL-^	same as "CTRL-W ^"
CTRL-W_CTRL-_	CTRL-W CTRL-_	same as "CTRL-W _"
CTRL-W_+	CTRL-W +	increase current window height N lines
CTRL-W_-	CTRL-W -	decrease current window height N lines
CTRL-W_<	CTRL-W <	decrease current window width N columns
CTRL-W_=	CTRL-W =	make all windows the same height & width
CTRL-W_>	CTRL-W >	increase current window width N columns
CTRL-W_H	CTRL-W H	move current window to the far left
CTRL-W_J	CTRL-W J	move current window to the very bottom
CTRL-W_K	CTRL-W K	move current window to the very top
CTRL-W_L	CTRL-W L	move current window to the far right
CTRL-W_P	CTRL-W P	go to preview window
CTRL-W_R	CTRL-W R	rotate windows upwards N times
CTRL-W_S	CTRL-W S	same as "CTRL-W s"
CTRL-W_T	CTRL-W T	move current window to a new tabpage
CTRL-W_W	CTRL-W W	go to N previous window (wrap around)
CTRL-W_]	CTRL-W]	split window and jump to tag under cursor
CTRL-W_^	CTRL-W ^	split current window and edit alternate file N
CTRL-W__	CTRL-W _	set current window height to N (default: very high)
CTRL-W_b	CTRL-W b	go to bottom window
CTRL-W_c	CTRL-W c	close current window (like :close)
CTRL-W_d	CTRL-W d	split window and jump to definition under the cursor
CTRL-W_f	CTRL-W f	split window and edit file name under the cursor
CTRL-W_F	CTRL-W F	split window and edit file name under the cursor and jump to the line number following the file name.CTRL-W_g_CTRL-] CTRL-W g CTRL-] split window and do :tjump to tag under cursor
CTRL-W_g]	CTRL-W g]	split window and do :tselect for tag under cursor
CTRL-W_g}	CTRL-W g }	do a :ptjump to the tag under the cursor
CTRL-W_gf	CTRL-W g f	edit file name under the cursor in a new tabpage
CTRL-W_gF	CTRL-W g F	edit file name under the cursor in a new tabpage and jump to the line number following the file name.
CTRL-W_gt	CTRL-W g t	same as gt: go to next tabpage
CTRL-W_gT	CTRL-W g T	same as gT: go to previous tabpage
CTRL-W_g<Tab>	CTRL-W g <Tab>	same as g<Tab>: go to last accessed tab page
CTRL-W_h	CTRL-W h	go to Nth left window (stop at first window)
CTRL-W_i	CTRL-W i	split window and jump to declaration of identifier under the cursor
CTRL-W_j	CTRL-W j	go N windows down (stop at last window)

Tag	Command	Normal-mode action
CTRL-W_k	CTRL-W k	go N windows up (stop at first window)
CTRL-W_l	CTRL-W l	go to Nth right window (stop at last window)
CTRL-W_n	CTRL-W n	open new window, N lines high
CTRL-W_o	CTRL-W o	close all but current window (like :only)
CTRL-W_p	CTRL-W p	go to previous (last accessed) window
CTRL-W_q	CTRL-W q	quit current window (like :quit)
CTRL-W_r	CTRL-W r	rotate windows downwards N times
CTRL-W_s	CTRL-W s	split current window in two parts, new window N lines high
CTRL-W_t	CTRL-W t	go to top window
CTRL-W_v	CTRL-W v	split current window vertically, new window N columns wide
CTRL-W_w	CTRL-W w	go to N next window (wrap around)
CTRL-W_x	CTRL-W x	exchange current window with window N (default: next window)
CTRL-W_z	CTRL-W z	close preview window
CTRL-W_bar	CTRL-W |	set window width to N columns
CTRL-W_}	CTRL-W }	show tag under cursor in preview window
CTRL-W_<Down>	CTRL-W <Down>	same as "CTRL-W j"
CTRL-W_<Up>	CTRL-W <Up>	same as "CTRL-W k"
CTRL-W_<Left>	CTRL-W <Left>	same as "CTRL-W h"
CTRL-W_<Right> CTRL-W <Right>	same as "CTRL-W l"	

2.3 Square bracket commands

[ ]

Tag	Char	Note Normal-mode action
[_CTRL-D	[CTRL-D	jump to first #define found in current and included files matching the word under the cursor, start searching at beginning of current file
[_CTRL-I	[CTRL-I	jump to first line in current and included files that contains the word under the cursor, start searching at beginning of current file
[#	[#	1 cursor to N previous unmatched #if, #else or #ifdef
['	['	1 cursor to previous lowercase mark, on first non-blank
[(	[(	1 cursor N times back to unmatched '('
[star	[*	1 same as "[/"
[`	[` 1 cursor to previous lowercase mark	
[/	[/	1 cursor to N previous start of a C comment
[C	[C	1 jump to N previous multicursor
[D	[D	list all defines found in current and included files matching the word under the cursor, start searching at beginning of current file
[I	[I	list all lines found in current and included files that contain the word under the cursor, start searching at beginning of current file
[P	[P	2 same as "[p"
[[	[[	1 cursor N sections backward
[]	[]	1 cursor N SECTIONS backward
[c	[c	1 cursor N times backwards to start of change
[d	[d	show first #define found in current and included files matching the word under the cursor, start searching at beginning of current file
[f	[f	same as "gf"
[i	[i	show first line found in current and included files that contains the word under the cursor, start searching at beginning of current file
[m	[m	1 cursor N times back to start of member function
[p	[p	2 like "P", but adjust indent to current line
[s	[s	1 move to the previous misspelled word
[z	[z	1 move to start of open fold
[{	[{	1 cursor N times back to unmatched '{'
[<MiddleMouse> [<MiddleMouse>	2 same as "[p"	
]_CTRL-D	] CTRL-D	jump to first #define found in current and included files matching the word under the cursor, start searching at cursor position
]_CTRL-I	] CTRL-I	jump to first line in current and included files that contains the word under the cursor, start searching at cursor position
]#	]#	1 cursor to N next unmatched #endif or #else
]'	]'	1 cursor to next lowercase mark, on first non-blank
])	])	1 cursor N times forward to unmatched ')'
]star	]*	1 same as "]/"
]`	]` 1 cursor to next lowercase mark	
]/	]/	1 cursor to N next end of a C comment
]C	]C	1 jump to N next multicursor
]D	]D	list all #defines found in current and included files matching the word under the cursor, start searching at cursor position
]I	]I	list all lines found in current and included files that contain the word under the cursor, start searching at cursor position
]P	]P	2 same as "[p"
][	][	1 cursor N SECTIONS forward
]]	]]	1 cursor N sections forward
]c	]c	1 cursor N times forward to start of change
]d	]d	show first #define found in current and included files matching the word under the cursor, start searching at cursor position
]f	]f	same as "gf"
]i	]i	show first line found in current and included files that contains the word under the cursor, start searching at cursor position
]m	]m	1 cursor N times forward to end of member function
]p	]p	2 like "p", but adjust indent to current line
]s	]s	1 move to next misspelled word
]z	]z	1 move to end of open fold

Tag	Char	Note Normal-mode action
]}	]}	1 cursor N times forward to unmatched '}'
]<MiddleMouse>]<MiddleMouse>	2 same as "]p"	

2.4 Commands starting with 'g'

g

Tag	Char	Note Normal-mode action
g_CTRL-A	g CTRL-A	insert an ascending number at each multicursor ("counter")
g_CTRL-G	g CTRL-G	show information about current cursor position
g_CTRL-H	g CTRL-H	start Select block mode
g_CTRL-]	g CTRL-]	:tjump to the tag under the cursor
g#	g#	1 like "#", but without using "\<" and "\>"
g$	g$	1 when 'wrap' off go to rightmost character of the current line that is on the screen; when 'wrap' on go to the rightmost character of the current screen line
g&	g&	2 repeat last ":s" on all lines
g'	g'{mark}	1 like ' but without changing the jumplist
g`	g`{mark}	1 like ` but without changing the jumplist
gstar	g*	1 like "*", but without using "\<" and "\>"
g+	g+	go to newer text state N times
g,	g,	1 go to N newer position in change list
g-	g-	go to older text state N times
g0	g0	1 when 'wrap' off go to leftmost character of the current line that is on the screen; when 'wrap' on go to the leftmost character of the current screen line
g8	g8	print hex value of bytes used in UTF-8 character under the cursor
g;	g;	1 go to N older position in change list
g<	g<	display previous command output
g?	g?	2 Rot13 encoding operator
g?g?	g??	2 Rot13 encode current line
g?g?	g?g?	2 Rot13 encode current line
gD	gD	1 go to definition of word under the cursor in current file
gE	gE	1 go backwards to the end of the previous WORD
gH	gH	start Select line mode
gI	gI	2 like "I", but always start in column 1
gJ	gJ	2 join lines without inserting space
gN	gN	1,2 find the previous match with the last used search pattern and Visually select it
gP	["x]gP	2 put the text [from register x] before the cursor N times, leave the cursor after it
gR	gR	2 enter Virtual Replace mode
gT	gT	go to the previous tabpage
gU	gU{motion}	2 make Nmove text uppercase
gV	gV	don't reselect the previous Visual area when executing a mapping or menu in Select mode
g]	g]	:tselect on the tag under the cursor
g^	g^	1 when 'wrap' off go to leftmost non-white character of the current line that is on the screen; when 'wrap' on go to the leftmost non-white character of the current screen line
g_	g_	1 cursor to the last CHAR N - 1 lines lower
ga	ga	print ascii value of character under the cursor
gd	gd	1 go to definition of word under the cursor in current function
ge	ge	1 go backwards to the end of the previous word
gf	gf	start editing the file whose name is under the cursor
gF	gF	start editing the file whose name is under the cursor and jump to the line number following the filename.
gg	gg	1 cursor to line N, default first line
gh	gh	start Select mode
gi	gi	2 like "i", but first move to the '^ mark
gj	gj	1 like "j", but when 'wrap' on go N screen lines down
gk	gk	1 like "k", but when 'wrap' on go N screen lines up
gm	gm	1 go to character at middle of the screenline
gM	gM	1 go to character at middle of the text line
gn	gn	1,2 find the next match with the last used search pattern and Visually select it
go	go	1 cursor to byte N in the buffer
gp	["x]gp	2 put the text [from register x] after the cursor N times, leave the cursor after it
gq	gq{motion}	2 format Nmove text
gr	gr{char}	2 virtual replace N chars with {char}
gs	gs	go to sleep for N seconds (default 1)
gt	gt	go to the next tabpage
gu	gu{motion}	2 make Nmove text lowercase
gv	gv	reselect the previous Visual area

Tag	Char	Note Normal-mode action
gw	gw{motion}	2 format Nmove text and keep cursor
gx	gx	execute application for filepath at cursor
g@	g@{motion}	call 'operatorfunc'
g~	g~{motion}	2 swap case for Nmove text
g<Down>	g<Down>	1 same as "gj"
g<End>	g<End>	1 same as "g$" but go to the rightmost non-blank character instead
g<Home>	g<Home>	1 same as "g0"
g<LeftMouse>	g<LeftMouse>	":tag" on the keyword at mouse-click
	g<MiddleMouse>	same as <C-MiddleMouse>
g<RightMouse>	g<RightMouse>	same as <C-RightMouse>
g<Tab>	g<Tab>	go to last accessed tabpage
g<Up>	g<Up>	1 same as "gk"

2.5 Commands starting with 'z'

z

Tag	Char	Note Normal-mode action
z<CR>	z<CR>	redraw, cursor line to top of window, cursor on first non-blank
zN<CR>	z{height}<CR>	redraw, make window {height} lines high
z+	z+	cursor on line N (default line below window), otherwise like "z<CR>"
z-	z-	redraw, cursor line at bottom of window, cursor on first non-blank
z.	z.	redraw, cursor line to center of window, cursor on first non-blank
z=	z=	give spelling suggestions
zA	zA	open a closed fold or close an open fold recursively
zC	zC	close folds recursively
zD	zD	delete folds recursively
zE	zE	eliminate all folds
zF	zF	create a fold for N lines
zG	zG	temporarily mark word as correctly spelled
zH	zH	when 'wrap' off scroll half a screenwidth to the right
zL	zL	when 'wrap' off scroll half a screenwidth to the left
zM	zM	set 'foldlevel' to zero
zN	zN	set 'foldenable'
zO	zO	open folds recursively
zR	zR	set 'foldlevel' to the deepest fold
zW	zW	temporarily mark word as incorrectly spelled
zX	zX	re-apply 'foldlevel'
z^	z^	cursor on line N (default line above window), otherwise like "z-"
za	za	open a closed fold, close an open fold
zb	zb	redraw, cursor line at bottom of window
zc	zc	close a fold
zd	zd	delete a fold
ze	ze	when 'wrap' off scroll horizontally to position the cursor at the end (right side) of the screen
zf	zf{motion}	create a fold for Nmove text
zg	zg	permanently mark word as correctly spelled
zh	zh	when 'wrap' off scroll screen N characters to the right
zi	zi	toggle 'foldenable'
zj	zj	1 move to the start of the next fold
zk	zk	1 move to the end of the previous fold
zl	zl	when 'wrap' off scroll screen N characters to the left
zm	zm	subtract one from 'foldlevel'
zn	zn	reset 'foldenable'
zo	zo	open fold
zp	zp	paste in block-mode without trailing spaces
zP	zP	paste in block-mode without trailing spaces
zr	zr	add one to 'foldlevel'
zs	zs	when 'wrap' off scroll horizontally to position the cursor at the start (left side) of the screen
zt	zt	redraw, cursor line at top of window
zuw	zuw	undo zw
zug	zug	undo zg
zuW	zuW	undo zW
zuG	zuG	undo zG
zv	zv	open enough folds to view the cursor line
zw	zw	permanently mark word as incorrectly spelled
zx	zx	re-apply 'foldlevel' and do "zv"
zy	zy	yank without trailing spaces
zz	zz	redraw, cursor line at center of window
z<Left>	z<Left>	same as "zh"
z<Right>	z<Right>	same as "zl"

2.6 Operator-pending mode

operator-pending-index

These can be used after an operator, but before a {motion} has been entered.

Tag	Char	Operator-pending-mode action
o_v	v	force operator to work charwise
o_V	V	force operator to work linewise
o_CTRL-V	CTRL-V	force operator to work blockwise

3. Visual mode

visual-index

Most commands in Visual mode are the same as in Normal mode. The ones listed
here are those that are different.

Tag	Command	Note Visual-mode action
v_CTRL-_CTRL-N CTRL-\ CTRL-N	stop Visual mode	
v_CTRL-_CTRL-G CTRL-\ CTRL-G	go to Normal mode	
v_CTRL-A	CTRL-A	2 add N to number in highlighted text
v_CTRL-C	CTRL-C	stop Visual mode
v_CTRL-G	CTRL-G	toggle between Visual mode and Select mode
v_<BS>	<BS>	2 Select mode: delete highlighted area
v_CTRL-H	CTRL-H	2 same as <BS>
v_CTRL-O	CTRL-O	switch from Select to Visual mode for one command
v_CTRL-V	CTRL-V	make Visual mode blockwise or stop Visual mode
v_CTRL-X	CTRL-X	2 subtract N from number in highlighted text
v_<Esc>	<Esc>	stop Visual mode
v_CTRL-]	CTRL-]	jump to highlighted tag
v_!	!{filter}	2 filter the highlighted lines through the external command {filter}
v_:	:	start a command-line with the highlighted lines as a range
v_<	<	2 shift the highlighted lines one 'shiftwidth' left
v_=	=	2 filter the highlighted lines through the external program given with the 'equalprg' option
v_>	>	2 shift the highlighted lines one 'shiftwidth' right
v_b_A	A	2 block mode: append same text in all lines, after the highlighted area
v_C	C	2 delete the highlighted lines and start insert
v_D	D	2 delete the highlighted lines
v_b_I	I	2 block mode: insert same text in all lines, before the highlighted area
v_J	J	2 join the highlighted lines
v_K	K	run 'keywordprg' on the highlighted area
v_O	O	move horizontally to other corner of area
v_P	P	replace highlighted area with register contents; registers are unchanged
v_R	R	2 delete the highlighted lines and start insert
v_S	S	2 delete the highlighted lines and start insert
v_U	U	2 make highlighted area uppercase
v_V	V	make Visual mode linewise or stop Visual mode
v_X	X	2 delete the highlighted lines
v_Y	Y	yank the highlighted lines
v_aquote	a"	extend highlighted area with a double quoted string
v_a'	a'	extend highlighted area with a single quoted string
v_a(	a(	same as ab
v_a)	a)	same as ab
v_a<	a<	extend highlighted area with a <> block
v_a>	a>	same as a<
v_aB	aB	extend highlighted area with a {} block
v_aW	aW	extend highlighted area with "a WORD"
v_a[	a[	extend highlighted area with a [] block
v_a]	a]	same as a[
v_a`	a`	extend highlighted area with a backtick quoted string
v_ab	ab	extend highlighted area with a () block
v_al	al	extend highlighted area to all lines
v_ap	ap	extend highlighted area with a paragraph
v_as	as	extend highlighted area with a sentence
v_at	at	extend highlighted area with a tag block
v_aw	aw	extend highlighted area with "a word"
v_a{	a{	same as aB
v_a}	a}	same as aB
v_c	c	2 delete highlighted area and start insert
v_d	d	2 delete highlighted area
v_g_CTRL-A	g CTRL-A	2 add N to number in highlighted text
v_g_CTRL-X	g CTRL-X	2 subtract N from number in highlighted text
v_gJ	gJ	2 join the highlighted lines without inserting spaces
v_gq	gq	2 format the highlighted lines
v_gv	gv	exchange current and previous highlighted area

Tag	Command	Note Visual-mode action
v_iquote	i"	extend highlighted area with a double quoted string (without quotes)
v_i'	i'	extend highlighted area with a single quoted string (without quotes)
v_i(	i(	same as ib
v_i)	i)	same as ib
v_i<	i<	extend highlighted area with inner <> block
v_i>	i>	same as i<
v_iB	iB	extend highlighted area with inner {} block
v_iW	iW	extend highlighted area with "inner WORD"
v_i[	i[	extend highlighted area with inner [] block
v_i]	i]	same as i[
v_i`	i`	extend highlighted area with a backtick quoted string (without the backticks)
v_ib	ib	extend highlighted area with inner () block
v_il	il	select inner line under cursor
v_ip	ip	extend highlighted area with inner paragraph
v_is	is	extend highlighted area with inner sentence
v_it	it	extend highlighted area with inner tag block
v_iw	iw	extend highlighted area with "inner word"
v_i{	i{	same as iB
v_i}	i}	same as iB
v_o	o	move cursor to other corner of area
v_p	p	replace highlighted area with register contents; deleted text in unnamed register
v_r	r	2 replace highlighted area with a character
v_s	s	2 delete highlighted area and start insert
v_u	u	2 make highlighted area lowercase
v_v	v	make Visual mode charwise or stop Visual mode
v_x	x	2 delete the highlighted area
v_y	y	yank the highlighted area
v_~	~	2 swap case for the highlighted area

4. Command-line editing

ex-edit-index

```
Get to the command-line with the ':', '!', '/' or '?' commands.
Normal characters are inserted at the current cursor position.
"Completion" below refers to context-sensitive completion.  It will complete
file names, tags, commands etc. as appropriate.
```

Tag	Command	Command-line editing-mode action
	CTRL-@	not used
c_CTRL-A	CTRL-A	do completion on the pattern in front of the cursor and insert all matches
c_CTRL-B	CTRL-B	cursor to begin of command-line
c_CTRL-C	CTRL-C	same as <Esc>
c_CTRL-D	CTRL-D	list completions that match the pattern in front of the cursor
c_CTRL-E	CTRL-E	cursor to end of command-line
'cedit'	CTRL-F	default value for 'cedit': opens the command-line window; otherwise not used
c_CTRL-G	CTRL-G	next match when 'incsearch' is active
c_<BS>	<BS>	delete the character in front of the cursor
c_digraph	{char1} <BS> {char2} enter digraph when 'digraph' is on	
c_CTRL-H	CTRL-H	same as <BS>
c_<Tab>	<Tab>	if 'wildchar' is <Tab>: Do completion on the pattern in front of the cursor
c_<S-Tab>	<S-Tab>	same as CTRL-P
c_wildchar	'wildchar'	Do completion on the pattern in front of the cursor (default: <Tab>)
c_CTRL-I	CTRL-I	same as <Tab>
c_<NL>	<NL>	same as <CR>
c_CTRL-J	CTRL-J	same as <CR>
c_CTRL-K	CTRL-K {char1} {char2} enter digraph	
c_CTRL-L	CTRL-L	do completion on the pattern in front of the cursor and insert the longest common part
c_<CR>	<CR>	execute entered command
c_CTRL-M	CTRL-M	same as <CR>
c_CTRL-N	CTRL-N	after using 'wildchar' with multiple matches: go to next match, otherwise: recall older command-line from history.
	CTRL-O	not used
c_CTRL-P	CTRL-P	after using 'wildchar' with multiple matches: go to previous match, otherwise: recall older command-line from history.
c_CTRL-Q	CTRL-Q	same as CTRL-V, unless it's used for terminal control flow
c_CTRL-R	CTRL-R {regname} insert the contents of a register or object under the cursor as if typedc_CTRL-R_CTRL-R CTRL-R CTRL-R {regname}c_CTRL-R_CTRL-O CTRL-R CTRL-O {regname} insert the contents of a register or object under the cursor literally	
	CTRL-S	not used, or used for terminal control flow
c_CTRL-T	CTRL-T	previous match when 'incsearch' is active
c_CTRL-U	CTRL-U	remove all characters between the cursor and beginning of command-line
c_CTRL-V	CTRL-V	insert next non-digit literally, insert three digit decimal number as a single byte.
c_CTRL-W	CTRL-W	delete the word in front of the cursor
	CTRL-X	not used (reserved for completion)
	CTRL-Y	copy (yank) modeless selection
	CTRL-Z	not used (reserved for suspend)
c_<Esc>	<Esc>	abandon command-line without executing it
c_CTRL-[	CTRL-[	same as <Esc>
c_CTRL-_CTRL-N CTRL-\ CTRL-N	go to Normal mode, abandon command-line	
c_CTRL-_CTRL-G CTRL-\ CTRL-G	go to Normal mode, abandon command-line	
	CTRL-\ a - d	reserved for extensions

Tag	Command	Command-line editing-mode action
c_CTRL-_e	CTRL-\ e {expr} replace the command line with the result of {expr}	
	CTRL-\ f - z	reserved for extensions
	CTRL-\ others	not used
c_CTRL-]	CTRL-]	trigger abbreviation
c_CTRL-^	CTRL-^	toggle use of :lmap mappings
c_<Del>	<Del>	delete the character under the cursor
c_<Left>	<Left>	cursor left
c_<S-Left>	<S-Left>	cursor one word left
c_<C-Left>	<C-Left>	cursor one word left
c_<Right>	<Right>	cursor right
c_<S-Right>	<S-Right>	cursor one word right
c_<C-Right>	<C-Right>	cursor one word right
c_<Up>	<Up>	recall previous command-line from history that matches pattern in front of the cursor
c_<S-Up>	<S-Up>	recall previous command-line from history
c_<Down>	<Down>	recall next command-line from history that matches pattern in front of the cursor
c_<S-Down>	<S-Down>	recall next command-line from history
c_<Home>	<Home>	cursor to start of command-line
c_<End>	<End>	cursor to end of command-line
c_<PageDown>	<PageDown>	same as <S-Down>
c_<PageUp>	<PageUp>	same as <S-Up>
c_<Insert>	<Insert>	toggle insert/overstrike mode
c_<LeftMouse>	<LeftMouse>	cursor at mouse click

commands in wildmenu mode (see 'wildmenu')

Char	Wildmenu action
<Up>	move up to parent
<Down>	move down to submenu
<Left>	select the previous match
<Right>	select the next match
<CR>	move into submenu when doing menu completion
CTRL-E	stop completion and go back to original text
CTRL-Y	accept selected match and stop completion
other	stop completion and insert the typed character

commands in wildmenu mode with 'wildoptions' set to "pum"

Char	Wildmenu action
<PageUp>	select a match several entries back
<PageDown>	select a match several entries forward

5. Terminal mode

terminal-mode-index

In a terminal buffer all keys except CTRL-\ are forwarded to the terminal
job. If CTRL-\ is pressed, the next key is forwarded unless it is CTRL-N
or CTRL-O.
Use CTRL-_CTRL-N to go to Normal mode.
Use t_CTRL-_CTRL-O to execute one normal mode command and then return
to terminal mode.

You found it, Arthur! holy-grail

6. EX commands

Ex-command ex-cmd-index :index

This is a brief but complete listing of all the ":" commands, without
mentioning any arguments. The optional part of the command name is inside [].
The commands are sorted on the non-optional part of their name.

Tag	Command	Action
::	:	nothing
:range	:{range}	go to last line in {range}
:!	:!	filter lines or execute an external command
:!!	:!!	repeat last ":!" command
:#	:#	same as ":number"
:&	:&	repeat last ":substitute"
:star	:*	use the last Visual area, like ":'<,'>"
:<	:<	shift lines one 'shiftwidth' left
:=	:=	print the last line number
:>	:>	shift lines one 'shiftwidth' right
:@	:@	execute contents of a register
:@@	:@@	repeat the previous ":@"
:2match	:2mat[ch]	define a second match to highlight
:3match	:3mat[ch]	define a third match to highlight
:Next	:N[ext]	go to previous file in the argument list
:append	:a[ppend]	append text
:abbreviate	:ab[breviate]	enter abbreviation
:abclear	:abc[lear]	remove all abbreviations
:aboveleft	:abo[veleft]	make split window appear left or above
:all	:al[l]	open a window for each file in the argument list
:amenu	:am[enu]	enter new menu item for all modes
:anoremenu	:an[oremenu]	enter a new menu for all modes that will not be remapped
:args	:ar[gs]	print the argument list
:argadd	:arga[dd]	add items to the argument list
:argdedupe	:argded[upe]	remove duplicates from the argument list
:argdelete	:argd[elete]	delete items from the argument list
:argedit	:arge[dit]	add item to the argument list and edit it
:argdo	:argdo	do a command on all items in the argument list
:argglobal	:argg[lobal]	define the global argument list
:arglocal	:argl[ocal]	define a local argument list
:argument	:argu[ment]	go to specific file in the argument list
:ascii	:as[cii]	print ascii value of character under the cursor
:autocmd	:au[tocmd]	enter or show autocommands
:augroup	:aug[roup]	select the autocommand group to use
:aunmenu	:aun[menu]	remove menu for all modes
:buffer	:b[uffer]	go to specific buffer in the buffer list
:bNext	:bN[ext]	go to previous buffer in the buffer list
:ball	:ba[ll]	open a window for each buffer in the buffer list
:badd	:bad[d]	add buffer to the buffer list
:balt	:balt	like ":badd" but also set the alternate file
:bcd	:bc[d]	change directory for buffer
:bchdir	:bch[dir]	change directory for buffer
:bdelete	:bd[elete]	remove a buffer from the buffer list
:belowright	:bel[owright]	make split window appear right or below
:bfirst	:bf[irst]	go to first buffer in the buffer list
:blast	:bl[ast]	go to last buffer in the buffer list
:bmodified	:bm[odified]	go to next buffer in the buffer list that has been modified
:bnext	:bn[ext]	go to next buffer in the buffer list
:botright	:bo[tright]	make split window appear at bottom or far right
:bprevious	:bp[revious]	go to previous buffer in the buffer list
:brewind	:br[ewind]	go to first buffer in the buffer list
:break	:brea[k]	break out of while loop
:breakadd	:breaka[dd]	add a debugger breakpoint
:breakdel	:breakd[el]	delete a debugger breakpoint
:breaklist	:breakl[ist]	list debugger breakpoints
:browse	:bro[wse]	use file selection dialog
:bufdo	:bufd[o]	execute command in each listed buffer
:buffers	:buffers	list all files in the buffer list
:bunload	:bun[load]	unload a specific buffer
:bwipeout	:bw[ipeout]	really delete a buffer

Tag	Command	Action
:change	:c[hange]	replace a line or series of lines
:cNext	:cN[ext]	go to previous error
:cNfile	:cNf[ile]	go to last error in previous file
:cabbrev	:ca[bbrev]	like ":abbreviate" but for Command-line mode
:cabclear	:cabc[lear]	clear all abbreviations for Command-line mode
:cabove	:cabo[ve]	go to error above current line
:caddbuffer	:cad[dbuffer]	add errors from buffer
:caddexpr	:cadde[xpr]	add errors from expr
:caddfile	:caddf[ile]	add error message to current quickfix list
:cafter	:caf[ter]	go to error after current cursor
:call	:cal[l]	call a function
:catch	:cat[ch]	part of a :try command
:cbefore	:cbe[fore]	go to error before current cursor
:cbelow	:cbel[ow]	go to error below current line
:cbottom	:cbo[ttom]	scroll to the bottom of the quickfix window
:cbuffer	:cb[uffer]	parse error messages and jump to first error
:cc	:cc	go to specific error
:cclose	:ccl[ose]	close quickfix window
:cd	:cd	change directory
:cdo	:cdo	execute command in each valid error list entry
:cfdo	:cfd[o]	execute command in each file in error list
:center	:ce[nter]	format lines at the center
:cexpr	:cex[pr]	read errors from expr and jump to first
:cfile	:cf[ile]	read file with error messages and jump to first
:cfirst	:cfir[st]	go to the specified error, default first one
:cgetbuffer	:cgetb[uffer]	get errors from buffer
:cgetexpr	:cgete[xpr]	get errors from expr
:cgetfile	:cg[etfile]	read file with error messages
:changes	:changes	print the change list
:chdir	:chd[ir]	change directory
:checkhealth	:che[ckhealth]	run healthchecks
:checkpath	:checkp[ath]	list included files
:checktime	:checkt[ime]	check timestamp of loaded buffers
:chistory	:chi[story]	list the error lists
:clast	:cla[st]	go to the specified error, default last one
:clearjumps	:cle[arjumps]	clear the jump list
:clist	:cl[ist]	list all errors
:close	:clo[se]	close current window
:cmap	:cm[ap]	like ":map" but for Command-line mode
:cmapclear	:cmapc[lear]	clear all mappings for Command-line mode
:cmenu	:cme[nu]	add menu for Command-line mode
:cnext	:cn[ext]	go to next error
:cnewer	:cnew[er]	go to newer error list
:cnfile	:cnf[ile]	go to first error in next file
:cnoremap	:cno[remap]	like ":noremap" but for Command-line mode
:cnoreabbrev	:cnorea[bbrev]	like ":noreabbrev" but for Command-line mode
:cnoremenu	:cnoreme[nu]	like ":noremenu" but for Command-line mode
:copy	:co[py]	copy lines
:colder	:col[der]	go to older error list
:colorscheme	:colo[rscheme]	load a specific color scheme
:command	:com[mand]	create user-defined command
:comclear	:comc[lear]	clear all user-defined commands
:compiler	:comp[iler]	do settings for a specific compiler
:continue	:con[tinue]	go back to :while
:confirm	:conf[irm]	prompt user when confirmation required
:const	:cons[t]	create a variable as a constant
:copen	:cope[n]	open quickfix window
:cprevious	:cp[revious]	go to previous error
:cpfile	:cpf[ile]	go to last error in previous file
:cquit	:cq[uit]	quit Vim with an error code

Tag	Command	Action
:crewind	:cr[ewind]	go to the specified error, default first one
:cunmap	:cu[nmap]	like ":unmap" but for Command-line mode
:cunabbrev	:cuna[bbrev]	like ":unabbrev" but for Command-line mode
:cunmenu	:cunme[nu]	remove menu for Command-line mode
:cwindow	:cw[indow]	open or close quickfix window
:delete	:d[elete]	delete lines
:debug	:deb[ug]	run a command in debugging mode
:debuggreedy	:debugg[reedy]	read debug mode commands from normal input
:defer	:defe[r]	call function when current function is done
:delcommand	:delc[ommand]	delete user-defined command
:delfunction	:delf[unction]	delete a user function
:delmarks	:delm[arks]	delete marks
:detach	:detach	detach the current UI
:diffupdate	:dif[fupdate]	update 'diff' buffers
:diffget	:diffg[et]	remove differences in current buffer
:diffoff	:diffo[ff]	switch off diff mode
:diffpatch	:diffp[atch]	apply a patch and show differences
:diffput	:diffpu[t]	remove differences in other buffer
:diffsplit	:diffs[plit]	show differences with another file
:diffthis	:difft[his]	make current window a diff window
:digraphs	:dig[raphs]	show or enter digraphs
:display	:di[splay]	display registers
:djump	:dj[ump]	jump to #define
:dl	:dl	short for :delete with the 'l' flag
:dlist	:dli[st]	list #defines
:doautocmd	:do[autocmd]	apply autocommands to current buffer
:doautoall	:doautoa[ll]	apply autocommands for all loaded buffers
:dp	:d[elete]p	short for :delete with the 'p' flag
:drop	:dr[op]	jump to window editing file or edit file in current window
:dsearch	:ds[earch]	list one #define
:dsplit	:dsp[lit]	split window and jump to #define
:edit	:e[dit]	edit a file
:earlier	:ea[rlier]	go to older change, undo
:echo	:ec[ho]	echoes the result of expressions
:echoerr	:echoe[rr]	like :echo, show like an error and use history
:echohl	:echoh[l]	set highlighting for echo commands
:echomsg	:echom[sg]	same as :echo, put message in history
:echon	:echon	same as :echo, but without <E0L>
:else	:el[se]	part of an :if command
:elseif	:elsei[f]	part of an :if command
:emenu	:em[enu]	execute a menu by name
:endif	:en[dif]	end previous :if
:endfor	:endfo[r]	end previous :for
:endfunction	:endf[unction]	end of a user function started with :function
:endtry	:endt[ry]	end previous :try
:endwhile	:endw[hile]	end previous :while
:enew	:ene[w]	edit a new, unnamed buffer
:eval	:ev[al]	evaluate an expression and discard the result
:ex	:ex	same as ":edit"
:execute	:exe[cute]	execute result of expressions
:exit	:exi[t]	same as ":xit"
:exmode	:exm[ode]	switch to Ex-mode
:exusage	:exu[sage]	overview of Ex commands
:fclose	:fc[lose]	close floating window
:file	:f[ile]	show or set the current file name
:files	:files	list all files in the buffer list
:filetype	:filet[ype]	switch file type detection on/off
:filter	:filt[er]	filter output of following command
:find	:fin[d]	find file in 'path' and edit it
:finally	:fina[lly]	part of a :try command

Tag	Command	Action
:finish	:fini[sh]	quit sourcing a Vim script
:first	:fir[st]	go to the first file in the argument list
:fold	:fo[ld]	create a fold
:foldclose	:foldc[lose]	close folds
:folddoopen	:foldd[oopen]	execute command on lines not in a closed fold
:folddoclosed	:folddoc[losed]	execute command on lines in a closed fold
:foldopen	:foldo[pen]	open folds
:for	:for	for loop
:function	:fu[nction]	define a user function
:global	:g[lobal]	execute commands for matching lines
:goto	:go[to]	go to byte in the buffer
:grep	:gr[ep]	run 'grepprg' and jump to first match
:grepadd	:grepa[dd]	like :grep, but append to current list
:gui	:gu[i]	start the GUI
:gvim	:gv[im]	start the GUI
:help	:h[elp]	open a help window
:helpclose	:helpc[lose]	close one help window
:helpgrep	:helpg[rep]	like ":grep" but searches help files
:helptags	:helpt[ags]	generate help tags for a directory
:highlight	:hi[ghlight]	specify highlighting methods
:hide	:hid[e]	hide current buffer for a command
:history	:his[tory]	print a history list
:horizontal	:hor[izontal]	following window command work horizontally
:insert	:i[nsert]	insert text
:iabbrev	:ia[bbrev]	like ":abbrev" but for Insert mode
:iabclear	:iabc[lear]	like ":abclear" but for Insert mode
:if	:if	execute commands when condition met
:ijump	:ij[ump]	jump to definition of identifier
:ilist	:il[ist]	list lines where identifier matches
:imap	:im[ap]	like ":map" but for Insert mode
:imapclear	:imapc[lear]	like ":mapclear" but for Insert mode
:imenu	:ime[nu]	add menu for Insert mode
:inoremap	:ino[remap]	like ":noremap" but for Insert mode
:inoreabbrev	:inorea[bbrev]	like ":noreabbrev" but for Insert mode
:inoremenu	:inoreme[nu]	like ":noremenu" but for Insert mode
:intro	:int[ro]	print the introductory message
:iput	:ip[ut]	like :put, but adjust the indent to the current line
:isearch	:is[earch]	list one line where identifier matches
:isplit	:isp[lit]	split window and jump to definition of identifier
:iunmap	:iu[nmap]	like ":unmap" but for Insert mode
:iunabbrev	:iuna[bbrev]	like ":unabbrev" but for Insert mode
:iunmenu	:iunme[nu]	remove menu for Insert mode
:join	:j[oin]	join lines
:jumps	:ju[mps]	print the jump list
:k	:k	set a mark
:keepalt	:keepa[lt]	following command keeps the alternate file
:keepmarks	:kee[pmarks]	following command keeps marks where they are
:keepjumps	:keepj[umps]	following command keeps jumplist and marks
:keeppatterns	:keepp[atterns]	following command keeps search pattern history
:lNext	:lN[ext]	go to previous entry in location list
:lNfile	:lNf[ile]	go to last entry in previous file
:list	:l[ist]	print lines
:labove	:lab[ove]	go to location above current line
:laddexpr	:lad[dexpr]	add locations from expr
:laddbuffer	:laddb[uffer]	add locations from buffer
:laddfile	:laddf[ile]	add locations to current location list
:lafter	:laf[ter]	go to location after current cursor
:last	:la[st]	go to the last file in the argument list
:language	:lan[guage]	set the language (locale)
:later	:lat[er]	go to newer change, redo

Tag	Command	Action
:lbefore	:lbe[fore]	go to location before current cursor
:lbelow	:lbel[ow]	go to location below current line
:lbottom	:lbo[ttom]	scroll to the bottom of the location window
:lbuffer	:lb[uffer]	parse locations and jump to first location
:lcd	:lc[d]	change directory locally
:lchdir	:lch[dir]	change directory locally
:lclose	:lcl[ose]	close location window
:ldo	:ld[o]	execute command in valid location list entries
:lfdo	:lfd[o]	execute command in each file in location list
:left	:le[ft]	left align lines
:leftabove	:lefta[bove]	make split window appear left or above
:let	:let	assign a value to a variable or option
:lexpr	:lex[pr]	read locations from expr and jump to first
:lfile	:lf[ile]	read file with locations and jump to first
:lfirst	:lfir[st]	go to the specified location, default first one
:lgetbuffer	:lgetb[uffer]	get locations from buffer
:lgetexpr	:lgete[xpr]	get locations from expr
:lgetfile	:lg[etfile]	read file with locations
:lgrep	:lgr[ep]	run 'grepprg' and jump to first match
:lgrepadd	:lgrepa[dd]	like :grep, but append to current list
:lhelpgrep	:lh[elpgrep]	like ":helpgrep" but uses location list
:lhistory	:lhi[story]	list the location lists
:ll	:ll	go to specific location
:llast	:lla[st]	go to the specified location, default last one
:llist	:lli[st]	list all locations
:lmake	:lmak[e]	execute external command 'makeprg' and parse error messages
:lmap	:lm[ap]	like ":map!" but includes Lang-Arg mode
:lmapclear	:lmapc[lear]	like ":mapclear!" but includes Lang-Arg mode
:lnext	:lne[xt]	go to next location
:lnewer	:lnew[er]	go to newer location list
:lnfile	:lnf[ile]	go to first location in next file
:lnoremap	:ln[oremap]	like ":noremap!" but includes Lang-Arg mode
:loadkeymap	:loadk[eymap]	load the following keymaps until EOF
:loadview	:lo[adview]	load view for current window from a file
:lockmarks	:loc[kmarks]	following command keeps marks where they are
:lockvar	:lockv[ar]	lock variables
:log	:log	view a log file
:lolder	:lol[der]	go to older location list
:lopen	:lop[en]	open location window
:lprevious	:lp[revious]	go to previous location
:lpfile	:lpf[ile]	go to last location in previous file
:lrewind	:lr[ewind]	go to the specified location, default first one
:ls	:ls	list all buffers
:lsp	:lsp	language server protocol
:ltag	:lt[ag]	jump to tag and add matching tags to the location list
:lunmap	:lu[nmap]	like ":unmap!" but includes Lang-Arg mode
:lua	:lua	execute Lua command
:luado	:luad[o]	execute Lua command for each line
:luafile	:luaf[ile]	execute Lua script file
:lvimgrep	:lv[imgrep]	search for pattern in files
:lvimgrepadd	:lvimgrepa[dd]	like :vimgrep, but append to current list
:lwindow	:lw[indow]	open or close location window
:move	:m[ove]	move lines
:mark	:ma[rk]	set a mark
:make	:mak[e]	execute external command 'makeprg' and parse error messages
:map	:map	show or enter a mapping
:mapclear	:mapc[lear]	clear all mappings for Normal and Visual mode
:marks	:marks	list all marks
:match	:mat[ch]	define a match to highlight
:menu	:me[nu]	enter a new menu item:menutranslate :menut[ranslate] add a menu translation item

Tag	Command	Action
:messages	:mes[sages]	view previously displayed messages
:mkexrc	:mk[exrc]	write current mappings and settings to a file
:mksession	:mks[ession]	write session info to a file
:mkspell	:mksp[ell]	produce .spl spell file
:mkvimrc	:mkv[imrc]	write current mappings and settings to a file
:mkview	:mkvie[w]	write view of current window to a file
:mode	:mod[e]	show or change the screen mode
:next	:n[ext]	go to next file in the argument list
:new	:new	create a new empty window
:nmap	:nm[ap]	like ":map" but for Normal mode
:nmapclear	:nmapc[lear]	clear all mappings for Normal mode
:nmenu	:nme[nu]	add menu for Normal mode
:nnoremap	:nn[oremap]	like ":noremap" but for Normal mode
:nnoremenu	:nnoreme[nu]	like ":noremenu" but for Normal mode
:noautocmd	:noa[utocmd]	following commands don't trigger autocommands
:noremap	:no[remap]	enter a mapping that will not be remapped
:nohlsearch	:noh[lsearch]	suspend 'hlsearch' highlighting
:noreabbrev	:norea[bbrev]	enter an abbreviation that will not be remapped
:noremenu	:noreme[nu]	enter a menu that will not be remapped
:normal	:norm[al]	execute Normal mode commands
:noswapfile	:nos[wapfile]	following commands don't create a swap file
:number	:nu[mber]	print lines with line number
:nunmap	:nun[map]	like ":unmap" but for Normal mode
:nunmenu	:nunme[nu]	remove menu for Normal mode
:oldfiles	:ol[dfiles]	list files that have marks in the shada file
:omap	:om[ap]	like ":map" but for Operator-pending mode
:omapclear	:omapc[lear]	remove all mappings for Operator-pending mode
:omenu	:ome[nu]	add menu for Operator-pending mode
:only	:on[ly]	close all windows except the current one
:onoremap	:ono[remap]	like ":noremap" but for Operator-pending mode
:onoremenu	:onoreme[nu]	like ":noremenu" but for Operator-pending mode
:options	:opt[ions]	open the options-window
:ounmap	:ou[nmap]	like ":unmap" but for Operator-pending mode
:ounmenu	:ounme[nu]	remove menu for Operator-pending mode
:packadd	:pa[ckadd]	add a plugin from 'packpath'
:packdel	:packd[el]	remove vim.pack plugins
:packloadall	:packl[oadall]	load all packages under 'packpath'
:packupdate	:packu[pdate]	update vim.pack plugins
:pbuffer	:pb[uffer]	edit buffer in the preview window
:pclose	:pc[lose]	close preview window
:pedit	:ped[it]	edit file in the preview window
:perl	:pe[rl]	execute perl command
:perldo	:perld[o]	execute perl command for each line
:perlfile	:perlf[ile]	execute perl script file
:print	:p[rint]	print lines
:profdel	:profd[el]	stop profiling a function or script
:profile	:prof[ile]	profiling functions and scripts
:pop	:po[p]	jump to older entry in tag stack
:popup	:popu[p]	popup a menu by name
:ppop	:pp[op]	":pop" in preview window
:preserve	:pre[serve]	write all text to swap file
:previous	:prev[ious]	go to previous file in argument list
:psearch	:ps[earch]	like ":ijump" but shows match in preview window
:ptag	:pt[ag]	show tag in preview window
:ptNext	:ptN[ext]	:tNext in preview window
:ptfirst	:ptf[irst]	:trewind in preview window
:ptjump	:ptj[ump]	:tjump and show tag in preview window
:ptlast	:ptl[ast]	:tlast in preview window
:ptnext	:ptn[ext]	:tnext in preview window
:ptprevious	:ptp[revious]	:tprevious in preview window

Tag	Command	Action
:ptrewind	:ptr[ewind]	:trewind in preview window
:ptselect	:pts[elect]	:tselect and show tag in preview window
:put	:pu[t]	insert contents of register in the text
:pwd	:pw[d]	print current directory
:py3	:py3	execute Python 3 command
:python3	:python3	same as :py3
:py3do	:py3d[o]	execute Python 3 command for each line
:py3file	:py3f[ile]	execute Python 3 script file
:python	:py[thon]	execute Python command
:pydo	:pyd[o]	execute Python command for each line
:pyfile	:pyf[ile]	execute Python script file
:pyx	:pyx	execute python_x command
:pythonx	:pythonx	same as :pyx
:pyxdo	:pyxd[o]	execute python_x command for each line
:pyxfile	:pyxf[ile]	execute python_x script file
:quit	:q[uit]	quit current window (when one window quit Vim)
:quitall	:quita[ll]	quit Vim
:qall	:qa[ll]	quit Vim
:read	:r[ead]	read file into the text
:recover	:rec[over]	recover a file from a swap file
:redo	:red[o]	redo one undone change
:redir	:redi[r]	redirect messages to a file or register
:redraw	:redr[aw]	force a redraw of the display
:redrawstatus	:redraws[tatus]	force a redraw of the status line(s) and window bar(s):redrawtabline :redrawt[abline] force a redraw of the tabline
:registers	:reg[isters]	display the contents of registers
:resize	:res[ize]	change current window height
:restart	:restart	restart the Nvim server
:retab	:ret[ab]	change tab size
:return	:retu[rn]	return from a user function
:rewind	:rew[ind]	go to the first file in the argument list
:right	:ri[ght]	right align text
:rightbelow	:rightb[elow]	make split window appear right or below
:rshada	:rsh[ada]	read from shada file
:ruby	:rub[y]	execute Ruby command
:rubydo	:rubyd[o]	execute Ruby command for each line
:rubyfile	:rubyf[ile]	execute Ruby script file
:rundo	:rund[o]	read undo information from a file
:runtime	:ru[ntime]	source vim scripts in 'runtimepath'
:substitute	:s[ubstitute]	find and replace text
:sNext	:sN[ext]	split window and go to previous file in argument list
:sandbox	:san[dbox]	execute a command in the sandbox
:sargument	:sa[rgument]	split window and go to specific file in argument list
:sall	:sal[l]	open a window for each file in argument list
:saveas	:sav[eas]	save file under another name.
:sbuffer	:sb[uffer]	split window and go to specific file in the buffer list
:sbNext	:sbN[ext]	split window and go to previous file in the buffer list
:sball	:sba[ll]	open a window for each file in the buffer list
:sbfirst	:sbf[irst]	split window and go to first file in the buffer list
:sblast	:sbl[ast]	split window and go to last file in buffer list
:sbmodified	:sbm[odified]	split window and go to modified file in the buffer list
:sbnext	:sbn[ext]	split window and go to next file in the buffer list
:sbprevious	:sbp[revious]	split window and go to previous file in the buffer list
:sbrewind	:sbr[ewind]	split window and go to first file in the buffer list
:scriptnames	:scr[iptnames]	list names of all sourced Vim scripts:scriptencoding :scripte[ncoding] encoding used in sourced Vim script
:set	:se[t]	show or set options
:setfiletype	:setf[iletype]	set 'filetype', unless it was set already
:setglobal	:setg[lobal]	show global values of options
:setlocal	:setl[ocal]	show or set options locally

Tag	Command	Action
:sfind	:sf[ind]	split current window and edit file in 'path'
:sfirst	:sfir[st]	split window and go to first file in the argument list
:sign	:sig[n]	manipulate signs
:silent	:sil[ent]	run a command silently
:sleep	:sl[eep]	do nothing for a few seconds
:sleep!	:sl[eep]!	do nothing for a few seconds, without the cursor visible
:slast	:sla[st]	split window and go to last file in the argument list
:smagic	:sm[agic]	:substitute with 'magic'
:smap	:smap	like ":map" but for Select mode
:smapclear	:smapc[lear]	remove all mappings for Select mode
:smenu	:sme[nu]	add menu for Select mode
:snext	:sn[ext]	split window and go to next file in the argument list
:snomagic	:sno[magic]	:substitute with 'nomagic'
:snoremap	:snor[emap]	like ":noremap" but for Select mode
:snoremenu	:snoreme[nu]	like ":noremenu" but for Select mode
:sort	:sor[t]	sort lines
:source	:so[urce]	read Vim or Ex commands from a file
:spelldump	:spelld[ump]	split window and fill with all correct words
:spellgood	:spe[llgood]	add good word for spelling
:spellinfo	:spelli[nfo]	show info about loaded spell files
:spellrare	:spellra[re]	add rare word for spelling
:spellrepall	:spellr[epall]	replace all bad words like last z=
:spellundo	:spellu[ndo]	remove good or bad word
:spellwrong	:spellw[rong]	add spelling mistake
:split	:sp[lit]	split current window
:sprevious	:spr[evious]	split window and go to previous file in the argument list
:srewind	:sre[wind]	split window and go to first file in the argument list
:stop	:st[op]	suspend the editor or escape to a shell
:stag	:sta[g]	split window and jump to a tag
:startinsert	:star[tinsert]	start Insert mode:startgreplace :startg[replace] start Virtual Replace mode
:startreplace	:startr[eplace]	start Replace mode
:stopinsert	:stopi[nsert]	stop Insert mode
:stjump	:stj[ump]	do ":tjump" and split window
:stselect	:sts[elect]	do ":tselect" and split window
:sunhide	:sun[hide]	same as ":unhide"
:sunmap	:sunm[ap]	like ":unmap" but for Select mode
:sunmenu	:sunme[nu]	remove menu for Select mode
:suspend	:sus[pend]	same as ":stop"
:sview	:sv[iew]	split window and edit file read-only
:swapname	:sw[apname]	show the name of the current swap file
:syntax	:sy[ntax]	syntax highlighting
:syntime	:synti[me]	measure syntax highlighting speed
:syncbind	:sync[bind]	sync scroll binding
:t	:t	same as ":copy"
:tNext	:tN[ext]	jump to previous matching tag
:tabNext	:tabN[ext]	go to previous tabpage
:tabclose	:tabc[lose]	close current tabpage
:tabdo	:tabd[o]	execute command in each tabpage
:tabedit	:tabe[dit]	edit a file in a new tabpage
:tabfind	:tabf[ind]	find file in 'path', edit it in a new tabpage
:tabfirst	:tabfir[st]	go to first tabpage
:tablast	:tabl[ast]	go to last tabpage
:tabmove	:tabm[ove]	move tabpage to other position
:tabnew	:tabnew	edit a file in a new tabpage
:tabnext	:tabn[ext]	go to next tabpage
:tabonly	:tabo[nly]	close all tabpages except the current one
:tabprevious	:tabp[revious]	go to previous tabpage
:tabrewind	:tabr[ewind]	go to first tabpage
:tabs	:tabs	list the tabpages and what they contain
:tab	:tab	create new tab when opening new window

Tag	Command	Action
:tag	:ta[g]	jump to tag
:tags	:tags	show the contents of the tag stack
:tcd	:tc[d]	change directory for tabpage
:tchdir	:tch[dir]	change directory for tabpage
:terminal	:te[rminal]	open a terminal buffer
:tfirst	:tf[irst]	jump to first matching tag
:throw	:th[row]	throw an exception
:tjump	:tj[ump]	like ":tselect", but jump directly when there is only one match
:tlast	:tl[ast]	jump to last matching tag
:tlmenu	:tlm[enu]	add menu for Terminal-mode
:tlnoremenu	:tln[oremenu]	like ":noremenu" but for Terminal-mode
:tlunmenu	:tlu[nmenu]	remove menu for Terminal-mode
:tmapclear	:tmapc[lear]	remove all mappings for Terminal-mode
:tmap	:tma[p]	like ":map" but for Terminal-mode
:tmenu	:tm[enu]	define menu tooltip
:tnext	:tn[ext]	jump to next matching tag
:tnoremap	:tno[remap]	like ":noremap" but for Terminal-mode
:topleft	:to[pleft]	make split window appear at top or far left
:tprevious	:tp[revious]	jump to previous matching tag
:trewind	:tr[ewind]	jump to first matching tag
:trust	:trust	add or remove file from trust database
:try	:try	execute commands, abort on error or exception
:tselect	:ts[elect]	list matching tags and select one
:tunmap	:tunma[p]	like ":unmap" but for Terminal-mode
:tunmenu	:tu[nmenu]	remove menu tooltip
:undo	:u[ndo]	undo last change(s)
:undojoin	:undoj[oin]	join next change with previous undo block
:undolist	:undol[ist]	list leafs of the undo tree
:unabbreviate	:una[bbreviate]	remove abbreviation
:unhide	:unh[ide]	open a window for each loaded file in the buffer list
:uniq	:uni[q]	uniq lines
:unlet	:unl[et]	delete variable
:unlockvar	:unlo[ckvar]	unlock variables
:unmap	:unm[ap]	remove mapping
:unmenu	:unme[nu]	remove menu
:unsilent	:uns[ilent]	run a command not silently
:update	:up[date]	write buffer if modified
:uptime	:upt[ime]	show Nvim's uptime
:vglobal	:v[global]	execute commands for not matching lines
:version	:ve[rsion]	print version number and other info
:verbose	:verb[ose]	execute command with 'verbose' set
:vertical	:vert[ical]	make following command split vertically
:vimgrep	:vim[grep]	search for pattern in files
:vimgrepadd	:vimgrepa[dd]	like :vimgrep, but append to current list
:visual	:vi[sual]	same as ":edit", but turns off "Ex" mode
:viusage	:viu[sage]	overview of Normal mode commands
:view	:vie[w]	edit a file read-only
:vmap	:vm[ap]	like ":map" but for Visual+Select mode
:vmapclear	:vmapc[lear]	remove all mappings for Visual+Select mode
:vmenu	:vme[nu]	add menu for Visual+Select mode
:vnew	:vne[w]	create a new empty window, vertically split
:vnoremap	:vn[oremap]	like ":noremap" but for Visual+Select mode
:vnoremenu	:vnoreme[nu]	like ":noremenu" but for Visual+Select mode
:vsplit	:vs[plit]	split current window vertically
:vunmap	:vu[nmap]	like ":unmap" but for Visual+Select mode
:vunmenu	:vunme[nu]	remove menu for Visual+Select mode
:windo	:wind[o]	execute command in each window
:write	:w[rite]	write to a file
:wNext	:wN[ext]	write to a file and go to previous file in argument list
:wall	:wa[ll]	write all (changed) buffers

Tag	Command	Action
:while	:wh[ile]	execute loop for as long as condition met
:winsize	:wi[nsize]	get or set window size (obsolete)
:wincmd	:winc[md]	execute a Window (CTRL-W) command
:winpos	:winp[os]	get or set window position
:wnext	:wn[ext]	write to a file and go to next file in argument list
:wprevious	:wp[revious]	write to a file and go to previous file in argument list
:wq	:wq	write to a file and quit window or Vim
:wqall	:wqa[ll]	write all changed buffers and quit Vim
:wshada	:wsh[ada]	write to ShaDa file
:wundo	:wu[ndo]	write undo information to a file
:xit	:x[it]	write if buffer changed and close window
:xall	:xa[ll]	same as ":wqall"
:xmapclear	:xmapc[lear]	remove all mappings for Visual mode
:xmap	:xm[ap]	like ":map" but for Visual mode
:xmenu	:xme[nu]	add menu for Visual mode
:xnoremap	:xn[oremap]	like ":noremap" but for Visual mode
:xnoremenu	:xnoreme[nu]	like ":noremenu" but for Visual mode
:xunmap	:xu[nmap]	like ":unmap" but for Visual mode
:xunmenu	:xunme[nu]	remove menu for Visual mode
:yank	:y[ank]	yank lines into a register
:z	:z	print some lines
:~	:~	repeat last ":substitute"

Nvim reference

Introduction intro

Vim is a text editor which includes most commands from the Unix program "Vi" and many new ones.

An overview of this manual can be found in the file "help.txt", help.txt. It can be accessed from within Vim with the <Help> or <F1> key and with the :help command (just type ":help", without the bars or quotes). The 'helpfile' option can be set to the name of the help file, in case it is not located in the default place. You can jump to subjects like with tags: Use CTRL-] to jump to a subject under the cursor, use CTRL-T to jump back.

pronounce

Vim is pronounced as one word, like Jim. So Nvim is "En-Vim", two syllables.

This manual is a reference for all Nvim editor and API features. It is not an introduction; instead for beginners, there is a hands-on tutor, lua-guide, and user-manual.

Resources resources

internet www distribution Nvim home page:

https://neovim.io/

book

There are many resources to learn Vi, Vim, and Nvim. We recommend:

- "Practical Vim" by Drew Neil. Acclaimed for its focus on quickly learning common editing tasks with Vim.
- "Modern Vim" by Drew Neil. Explores new features in Nvim and Vim 8.
- http://vimcasts.org
- "Vim - Vi Improved" by Steve Oualline. This was the first book dedicated to Vim. Parts of it were included in the Vim user manual. frombook ISBN: 0735710015
- For more information see https://www.vim.org/iccf/click5.html
- Vim FAQ: https://vimhelp.org/vim_faq.txt.html

bugs bug-report feature-request Report bugs and request features here: https://github.com/neovim/neovim/issues Be brief, yet complete. Always give a reproducible example and try to find out which settings or other things trigger the bug. If Nvim crashed, try to get a backtrace (see dev-tools-backtrace).

Installing Nvim install

download upgrade ubuntu To install or upgrade Nvim, you can...
- Download a pre-built archive: https://github.com/neovim/neovim/releases
- Use your system package manager: https://github.com/neovim/neovim/blob/master/INSTALL.md#install-from-package
- Build from source: https://github.com/neovim/neovim/blob/master/INSTALL.md#install-from-source

Un-installing Nvim uninstall

To uninstall Nvim:
- If you downloaded a pre-built archive or built Nvim from source (e.g. `make install`), just delete its files, typically located in:

```
/usr/local/bin/nvim
/usr/local/share/nvim
```

  - To find where Nvim is installed, run these commands:

```
:echo v:progpath
:echo $VIMRUNTIME
```

- If you installed via package manager, read your package manager's documentation. Common examples:
  - APT (Debian, Ubuntu, ...): `apt-get remove neovim`
  - Homebrew (macOS): `brew uninstall neovim`
  - Scoop (Windows): `scoop uninstall neovim`

Sponsor Vim/Nvim development sponsor

Fixing bugs and adding new features takes a lot of time and effort. To show your appreciation for the work and motivate developers to continue working on Vim please send a donation.

The money you donated will be mainly used to help children in Uganda. See uganda. But at the same time donations increase the development team motivation to keep working on Vim!

For the most recent information about sponsoring look on the Vim web site:

https://www.vim.org/sponsor/

Nvim development is funded separately from Vim:

https://neovim.io/#sponsor

Bram Moolenaar Bram Moolenaar Bram-Moolenaar brammool

Nvim is a fork of the Vim ("Vi IMproved") text editor, which was originally developed by Bram Moolenaar. Searching his name within the source code of Nvim will reveal just how much of his work still remains in Nvim. On August 3, 2023, he passed away at the age of 62. If Vim or Nvim have been of use to you in your life, please read Uganda and consider honoring his memory however you may see fit.

- Obituary Articles: https://github.com/vim/vim/discussions/12742
- Say Farewell: https://github.com/vim/vim/discussions/12737

Keycodes

key-notation key-codes keycodes

These names for keys are used in the documentation. They can also be used with the ":map" command.

notation meaning equivalent decimal value(s)
<Nul> Zero CTRL-@ 0 (stored as 10) <Nul> <BS> Backspace CTRL-H 8 backspace <Tab> Tab CTRL-I 9 tab Tab linefeed
<NL> Linefeed CTRL-J 10 (used for <Nul>) <CR> Carriage return CTRL-M 13 carriage-return <Return> Same as <CR> <Return>
<Enter> Same as <CR> <Enter>
<Esc> Escape CTRL-[27 escape <Esc> <Space> Space 32 space <lt> Less-than < 60 <lt> <Bslash> Backslash \ 92 backslash <Bslash>
<Bar> Vertical bar | 124 <Bar> <Del> Delete 127 <CSI> Command sequence intro ALT-Esc 155 <CSI>

<EOL> End-of-line (can be <CR>, <NL> or <CR><NL>, Depends on system and 'fileformat') <EOL> <Ignore> Cancel wait-for-character <Ignore>
<NOP> Do nothing (no-op). Useful in mappings. <Nop> <Ignore> is a key, <NOP> is "absence of a key".

<Up> Cursor-up cursor-up cursor_up <Down> Cursor-down cursor-down cursor_down <Left> Cursor-left cursor-left cursor_left <Right> Cursor-right cursor-right cursor_right <S-Up> Shift-cursor-up <S-Down> Shift-cursor-down <S-Left> Shift-cursor-left <S-Right> Shift-cursor-right <C-Left> Control-cursor-left <C-Right> Control-cursor-right <F1> - <F12> Function keys 1 to 12 function_key function-key <S-F1> - <S-F12> Shift-function keys 1 to 12 <S-F1> <Help> Help key <Undo> Undo key <Find> Find key <Select> Select key <Insert> Insert key <Home> Home home
<End> End end
<PageUp> Page-up page_up page-up <PageDown> Page-down page_down page-down <kUp> Keypad cursor-up keypad-cursor-up
<kDown> Keypad cursor-down keypad-cursor-down
<kLeft> Keypad cursor-left keypad-cursor-left
<kRight> Keypad cursor-right keypad-cursor-right
<kHome> Keypad home (upper left) keypad-home <kEnd> Keypad end (lower left) keypad-end
<kOrigin> Keypad origin (middle) keypad-origin
<kPageUp> Keypad page-up (upper right) keypad-page-up <kPageDown> Keypad page-down (lower right) keypad-page-down <kDel> Keypad delete keypad-delete
<kPlus> Keypad + keypad-plus
<kMinus> Keypad - keypad-minus
<kMultiply> Keypad * keypad-multiply
<kDivide> Keypad / keypad-divide
<kPoint> Keypad . keypad-point
<kComma> Keypad , keypad-comma
<kEqual> Keypad = keypad-equal
<kEnter> Keypad Enter keypad-enter
<k0> - <k9> Keypad 0 to 9 keypad-0 keypad-9 <S-…> Shift-key shift <S- <C-…> Control-key control ctrl <C- <M-…> Alt-key or meta-key META ALT <M- <A-…> Same as <M-…> <A-
<T-…> Meta-key, when it's not alt <T- <D-…> Command-key or "super" key <D-

Note:

- Availability of some keys (<Help>, <S-Right>, ...) depends on the UI or host terminal.
- If numlock is on the TUI receives plain ASCII values, so mapping <k0>, <k1>, ..., <k9> and <kPoint> will not work.
- Nvim supports mapping multibyte chars with modifiers such as <M-ä>. Which combinations actually work depends on the UI or host terminal.
- When a key is pressed using a meta or alt modifier and no mapping exists for that keypress, Nvim may behave as though <Esc> was pressed before the key.
- It is possible to notate combined modifiers (e.g. <M-C-T> for CTRL-ALT-T), but your terminal must encode the input for that to work. tui-input

<>

Examples are often given in the <> notation. Sometimes this is just to make clear what you need to type, but often it can be typed literally, e.g., with the ":map" command. The rules are:

1. Printable characters are typed directly, except backslash and "<"
2. Backslash is represented with "\\", double backslash, or "<Bslash>".
3. Literal "<" is represented with "\<" or "<lt>". When there is no confusion possible, "<" can be used directly.
4. "<key>" means the special key typed (see the table above). Examples:
   - <Esc> Escape key
   - <C-G> CTRL-G
   - <Up> cursor up key
   - <C-LeftMouse> Control- left mouse click
   - <S-F11> Shifted function key 11
   - <M-a> Meta- a ('a' with bit 8 set)
   - <M-A> Meta- A ('A' with bit 8 set)

The <> notation uses <lt> to escape the special meaning of key names. Using a backslash also works, but only when 'cpoptions' does not include the 'B' flag.

Examples for mapping CTRL-H to the six characters "<Home>":

```
:imap <C-H> \<Home>
:imap <C-H> <lt>Home>
```

The first one only works when the 'B' flag is not in 'cpoptions'. The second one always works. To get a literal "<lt>" in a mapping:

```
:map <C-L> <lt>lt>
```

The notation can be used in a double quoted strings, using "\<" at the start, e.g. "\<C-Space>". This results in a special key code. To convert this back to readable text use keytrans().

Modes, introduction

vim-modes-intro vim-modes

Vim has seven BASIC modes:

Normal Normal-mode command-mode

- Normal mode: In Normal mode you can enter all the normal editor commands. If you start the editor you are in this mode. This is also known as command mode.
- Visual mode: This is like Normal mode, but the movement commands extend a highlighted area. When a non-movement command is used, it is executed for the highlighted area. See Visual-mode. If the 'showmode' option is on "-- VISUAL --" is shown at the bottom of the window.
- Select mode: This looks most like the MS-Windows selection mode. Typing a printable character deletes the selection and starts Insert mode. See Select-mode. If the 'showmode' option is on "-- SELECT --" is shown at the bottom of the window.
- Insert mode: In Insert mode the text you type is inserted into the buffer. See Insert-mode. If the 'showmode' option is on "-- INSERT --" is shown at the bottom of the window.
- Cmdline mode: In Command-line mode (also called Cmdline mode) you can enter one line of text at the bottom of the window. This is for the Ex commands, ":", the pattern search commands, "?" and "/", and the filter command, "!". Cmdline-mode
- Ex mode: Like Command-line mode, but after entering a command you remain in Ex-mode.

Terminal-mode

- Terminal mode: In Terminal mode all input (except CTRL-\) is sent to the process running in the current terminal buffer. If CTRL-\ is pressed, the next key is sent unless it is CTRL-N (CTRL-_CTRL-N) or CTRL-O (t_CTRL-_CTRL-O). If the 'showmode' option is on "-- TERMINAL --" is shown at the bottom of the window.

There are six ADDITIONAL modes. These are variants of the BASIC modes:

Operator-pending Operator-pending-mode

- Operator-pending mode: This is like Normal mode, but after an operator command has started, and Vim is waiting for a {motion} to specify the text that the operator will work on.
- Replace mode: Replace mode is a special case of Insert mode. You can do the same things as in Insert mode, but for each character you enter, one character of the existing text is deleted. See Replace-mode. If the 'showmode' option is on "-- REPLACE --" is shown at the bottom of the window.
- Virtual Replace mode: Virtual Replace mode is similar to Replace mode, but instead of file characters you are replacing screen real estate. See Virtual-Replace-mode. If the 'showmode' option is on "-- VREPLACE --" is shown at the bottom of the window.
- Insert Normal mode: Entered when CTRL-O is typed in Insert mode (see i_CTRL-O). This is like Normal mode, but after executing one command Vim returns to Insert mode. If the 'showmode' option is on "-- (insert) --" is shown at the bottom of the window.
- Insert Visual mode: Entered when starting a Visual selection from Insert mode, e.g., by using CTRL-O and then "v", "V" or CTRL-V. When the Visual selection ends, Vim returns to Insert mode. If the 'showmode' option is on "-- (insert) VISUAL --" is shown at the bottom of the window.
- Insert Select mode: Entered when starting Select mode from Insert mode. E.g., by dragging the mouse or <S-Right>. When the Select mode ends, Vim returns to Insert mode. If the 'showmode' option is on "-- (insert) SELECT --" is shown at the bottom of the window.

Switching from mode to mode

mode-switching

If for any reason you do not know which mode you are in, you can always get back to Normal mode by typing <Esc> twice. This doesn't work for Ex mode though, use ":visual". You will know you are back in Normal mode when you see the screen flash or hear the bell after you type <Esc>. However, when pressing <Esc> after using CTRL-O in Insert mode you get a beep but you are still in Insert mode, type <Esc> again.

i_esc

```
FROM mode      TO mode
               Normal  Visual  Select  Insert    Replace   Cmd-line  Ex >
Normal                 v V ^V    *4     *1        R gR     : / ? !   1q:
Visual          *2                ^G     c C        --        :       --
Select          *5     ^O ^G             *6         --        --      --
Insert          <Esc>    --      --               <Insert>    --      --
Replace         <Esc>    --      --    <Insert>               --      --
Command-line    *3       --      --     :start     --                --
Ex              :vi      --      --     --         --        --

-- not possible
```

- 1 Go from Normal mode to Insert mode by giving the command "i", "I", "a", "A", "o", "O", "c", "C", "s" or S".
- 2 Go from Visual mode to Normal mode by giving a non-movement command, which causes the command to be executed, or by hitting <Esc> "v", "V" or "CTRL-V" (see v_v), which just stops Visual mode without side effects.
- 3 Go from Command-line mode to Normal mode by:
  - Hitting <CR> or <NL>, which causes the entered command to be executed.
  - Deleting the complete line (e.g., with CTRL-U) and giving a final <BS>.
  - Hitting CTRL-C or <Esc>, which quits the command-line without executing the command. In the last case <Esc> may be the character defined with the 'wildchar' option, in which case it will start command-line completion. You can ignore that and type <Esc> again.
- 4 Go from Normal to Select mode by:
  - use the mouse to select text while 'selectmode' contains "mouse"
  - use a non-printable command to move the cursor while keeping the Shift key pressed, and the 'selectmode' option contains "key"
  - use "v", "V" or "CTRL-V" while 'selectmode' contains "cmd"
  - use "gh", "gH" or "g CTRL-H" g_CTRL-H
- 5 Go from Select mode to Normal mode by using a non-printable command to move the cursor, without keeping the Shift key pressed.
- 6 Go from Select mode to Insert mode by typing a printable character. The selection is deleted and the character is inserted.

CTRL-_CTRL-N i_CTRL-_CTRL-N c_CTRL-_CTRL-N v_CTRL-_CTRL-N t_CTRL-_CTRL-N Additionally the command CTRL-\ CTRL-N or <C-\><C-N> can be used to go to Normal mode from any other mode. This can be used to make sure Vim is in Normal mode, without causing a beep like <Esc> would. When used after a command that takes an argument, such as f or m, the timeout set with 'ttimeoutlen' applies.

CTRL-_CTRL-G i_CTRL-_CTRL-G c_CTRL-_CTRL-G v_CTRL-_CTRL-G CTRL-\ CTRL-G works the same as CTRL-_CTRL-N for backward compatibility.

:exm :exmode mode-Ex Ex-mode Ex EX E501 :exm[ode] [count]q: Switch to Ex mode. This is like the cmdwin except it stays open, in Insert-mode, so can you keep entering ":" commands (REPL style). Use :visual to exit this mode.

If a command moves the cursor or changes the buffer, the current line is printed. An empty line moves to (and prints) the next line.

Without [count], "q:" opens the regular cmdwin.

Implementation: a persistent cmdwin where <Enter> executes the current line as a command (against the window from which Ex mode was entered). Command output is presented as comment (") lines.

Window contents **window-contents**

In Normal mode and Insert/Replace mode the screen window will show the current contents of the buffer: What You See Is What You Get. There are two exceptions:

- When the 'cpoptions' option contains '$', and the change is within one line, the text is not directly deleted, but a '$' is put at the last deleted character.
- When inserting text in one window, other windows on the same text are not updated until the insert is finished.

Lines longer than the window width will wrap, unless the 'wrap' option is off (see below). The 'linebreak' option can be set to wrap at a blank character.

If the window has room after the last line of the buffer, Vim will show '~' in the first column of the last lines in the window, like this:

```
+----------------------+
|some line             |
|last line             |
|~                     |
|~                     |
+----------------------+
```

Thus the '~' lines indicate that the end of the buffer was reached.

If the last line in a window doesn't fit, Vim will indicate this with a '@' in the first column of the last lines in the window, like this:

```
+----------------------+
|first line            |
|second line           |
|@                     |
|@                     |
+----------------------+
```

Thus the '@' lines indicate that there is a line that doesn't fit in the window.

When the "lastline" flag is present in the 'display' option, you will not see '@' characters at the left side of window. If the last line doesn't fit completely, only the part that fits is shown, and the last three characters of the last line are replaced with "@@@", like this:

```
+----------------------+
|first line            |
|second line           |
|a very long line that d|
|oesn't fit in the wi@@@|
+----------------------+
```

If there is a single line that is too long to fit in the window, this is a special situation. Vim will show only part of the line, around where the cursor is. There are no special characters shown, so that you can edit all parts of this line.

The hl-NonText highlight group can be used to set special highlighting for the '@' and '~' characters. This makes it possible to distinguish them from real characters in the buffer.

The 'showbreak' option contains the string to put in front of wrapped lines.

wrap-off

If the 'wrap' option is off, long lines will not wrap. Only the part that fits on the screen is shown. If the cursor is moved to a part of the line that is not shown, the screen is scrolled horizontally. The advantage of this method is that columns are shown as they are and lines that cannot fit on the screen can be edited. The disadvantage is that you cannot see all the characters of a line at once. The 'sidescroll' option can be set to the minimal number of columns to scroll.

All normal ASCII characters are displayed directly on the screen. The <Tab> is replaced with the number of spaces that it represents. Other non-printing characters are replaced with "^{char}", where {char} is the non-printing character with 64 added. Thus character 7 (bell) will be shown as "^G". Characters between 127 and 160 are replaced with "~{char}", where {char} is the character with 64 subtracted. These characters occupy more than one position on the screen. The cursor can only be positioned on the first one.

If you set the 'number' option, all lines will be preceded with their number. Tip: If you don't like wrapping lines to mix with the line numbers, set the 'showbreak' option to eight spaces:

```
":set showbreak=\ \ \ \ \ \ \ "
```

If you set the 'list' option, <Tab> characters will not be shown as several spaces, but as "^I". A '$' will be placed at the end of the line, so you can find trailing blanks.

In Command-line mode only the command-line itself is shown correctly. The display of the buffer contents is updated as soon as you go back to Command mode.

The last line of the window is used for status and other messages. The status messages will only be used if an option is on:

```
status message               option       default    Unix default
current mode                 'showmode'       on          on
command characters           'showcmd'        on          off
cursor position              'ruler'          off         off
```

The current mode is "-- INSERT --" or "-- REPLACE --", see 'showmode'. The command characters are those that you typed but were not used yet.

If you have a slow terminal you can switch off the status messages to speed up editing:

```
:set nosc noru nosm
```

If there is an error, an error message will be shown for at least one second (in reverse video).

Some commands show how many lines were affected. Above which threshold this happens can be controlled with the 'report' option (default 2).

The name Vim and the full name of the current file name will be shown in the title bar. When the window is resized, Vim will automatically redraw the window. You may make the window as small as you like, but if it gets too small not a single line will fit in it. Make it at least 40 characters wide to be able to read most messages on the last line.

Definitions

definitions jargon

- buffer: Contains lines of text, usually from a file.
- screen: The whole area that Nvim uses to display things.
- window: A view on a buffer. There can be multiple windows for one buffer.
- frame: Windows are kept in a tree of frames. Each frame contains a column, row, or window ("leaf" frame).

A screen contains one or more windows, separated by status lines and with the command line at the bottom.

```
       +-------------------------------+
screen | window 1      | window 2      |
       |               |               |
       |               |               |
       |= status line =|= status line =|
       | window 3                      |
       |                               |
       |                               |
       |==== status line ==============|
       |command line                   |
       +-------------------------------+
```

The command line is also used for messages. It scrolls up the screen when there is not enough room in the command line.

A difference is made between four types of lines:

- buffer lines: The lines in the buffer. This is the same as the lines as they are read from/written to a file. They can be thousands of characters long.
- logical lines: The buffer lines with folding applied. Buffer lines in a closed fold are changed to a single logical line: "+-- 99 lines folded". They can be thousands of characters long.
- window lines: The lines displayed in a window: A range of logical lines with wrapping, line breaks, etc. applied. They can only be as long as the width of the window allows, longer lines are wrapped or truncated.
- screen lines: The lines of the screen that Nvim uses. Consists of the window lines of all windows, with status lines and the command line added. They can only be as long as the width of the screen allows. When the command line gets longer it wraps and lines are scrolled to make room.

```
buffer lines    logical lines   window lines    screen lines
-------------------------------------------------------------------------
1. one          1. one          1. +-- folded   1.  +-- folded
2. two          2. +-- folded   2. five         2.  five
3. three        3. five         3. six          3.  six
4. four         4. six          4. seven        4.  seven
5. five         5. seven                        5.  === status line ===
6. six                                          6.  aaa
7. seven                                        7.  bbb
                                                8.  ccc ccc c
1. aaa          1. aaa          1. aaa          9.  cc
2. bbb          2. bbb          2. bbb          10. ddd
3. ccc ccc ccc  3. ccc ccc ccc  3. ccc ccc c    11. ~
4. ddd          4. ddd          4. cc           12. === status line ===
                                5. ddd          13. (command line)
                                6. ~
```

API client
All external UIs and remote plugins (as opposed to regular Vim plugins) are "clients" in general; but we call something an "API client" if its purpose is to abstract or wrap the RPC API for the convenience of other applications (just like a REST client or SDK such as boto3 for AWS: you can speak AWS REST using an HTTP client like curl, but boto3 wraps that in a convenient python interface). For example, the Nvim node-client is an API client: https://github.com/neovim/node-client

Host
A plugin "host" is both a client (of the Nvim API) and a server (of an external platform, e.g. python). It is a remote plugin that hosts other plugins.

Remote plugin
Arbitrary code registered via :UpdateRemotePlugins, that runs in a separate process and communicates with Nvim via the api.

Nvim job control job-control E901 E903 E947 E948

Job control is a way to perform multitasking in Nvim, so scripts can spawn and control multiple processes without blocking the current Nvim instance.

Concepts

Job Id job-id

Each job is identified by an integer id, unique for the life of the current Nvim session. Each job-id is a valid channel-id: they share the same "key space". Functions like jobstart() return job ids; functions like jobstop(), chansend(), rpcnotify(), and rpcrequest() take job ids.

Job stdio streams form a channel which can send and receive raw bytes or msgpack-rpc messages.

Usage job-control-usage

To control jobs, use the "job..." family of functions: jobstart(), jobstop(), etc.

Example:

```
function! s:OnEvent(job_id, data, event) dict
  if a:event == 'stdout'
    let str = self.shell.' stdout: '.join(a:data)
  elseif a:event == 'stderr'
    let str = self.shell.' stderr: '.join(a:data)
  else
    let str = self.shell.' exited'
  endif

  call append(line('$'), str)
endfunction
let s:callbacks = {
\ 'on_stdout': function('s:OnEvent'),
\ 'on_stderr': function('s:OnEvent'),
\ 'on_exit': function('s:OnEvent')
\ }
let job1 = jobstart(['bash'], extend({'shell': 'shell 1'}, s:callbacks))
let job2 = jobstart(['bash', '-c', 'for i in {1..10}; do echo hello $i!; sleep 1; done'], extend({'shell': 'shell 2'}, s:callbacks))
```

To test the above script, copy it to a file ~/foo.vim and run it:

```
nvim -u ~/foo.vim
```

Description of what happens:
- Two bash shells are spawned by jobstart() with their stdin/stdout/stderr streams connected to nvim.
- The first shell is idle, waiting to read commands from its stdin.
- The second shell is started with -c which executes the command (a for-loop printing 0 through 9) and then exits.
- `OnEvent()` callback is passed to jobstart() to handle various job events. It displays stdout/stderr data received from the shells.

For on_stdout and on_stderr see channel-callback. on_exit
Arguments passed to on_exit callback: 0: job-id 1: Exit-code of the process, or 128+SIGNUM if by signal (e.g. 143 on SIGTERM). 2: Event type: "exit"

Note: Buffered stdout/stderr data which has not been flushed by the sender will not trigger the on_stdout/on_stderr callback (but if the process ends, the on_exit callback will be invoked). For example, "ruby -e" buffers output, so small strings will be buffered unless "auto-flushing" ($stdout.sync=true) is enabled.

```
function! Receive(job_id, data, event)
  echom printf('%s: %s',a:event,string(a:data))
endfunction
call jobstart(['ruby', '-e',
  \ '$stdout.sync = true; 5.times do sleep 1 and puts "Hello Ruby!" end'],
  \ {'on_stdout': 'Receive'})
```

https://github.com/neovim/neovim/issues/1592

Note: Job event handlers may receive partial (incomplete) lines. For a given invocation of on_stdout/on_stderr, `a:data` is not guaranteed to end with a newline.
- `abcdefg` may arrive as `['abc']`, `['defg']`.
- `abc\nefg` may arrive as `['abc', '']`, `['efg']` or `['abc']`, `['','efg']`, or even `['ab']`, `['c','efg']`. Easy way to deal with this: initialize a list as `['']`, then append to it as follows:

```
let s:chunks = ['']
func! s:on_stdout(job_id, data, event) dict
  let s:chunks[-1] .= a:data[0]
  call extend(s:chunks, a:data[1:])
endf
```

The jobstart-options dictionary is passed as self to the callback. The above example could be written in this "object-oriented" style:

```
let Shell = {}

function Shell.on_stdout(_job_id, data, event)
  call append(line('$'),
        \ printf('[%s] %s: %s', a:event, self.name, join(a:data[:-2])))
endfunction

let Shell.on_stderr = function(Shell.on_stdout)

function Shell.on_exit(job_id, _data, event)
  let msg = printf('job %d ("%s") finished', a:job_id, self.name)
  call append(line('$'), printf('[%s] BOOM!', a:event))
  call append(line('$'), printf('[%s] %s!', a:event, msg))
endfunction

function Shell.new(name, cmd)
  let object = extend(copy(g:Shell), {'name': a:name})
  let object.cmd = ['sh', '-c', a:cmd]
  let object.id = jobstart(object.cmd, object)
  $
  return object
endfunction

let instance = Shell.new('bomb',
      \ 'for i in $(seq 9 -1 1); do echo $i 1>&$((i % 2 + 1)); sleep 1; done')
```

To send data to the job's stdin, use chansend():

```
:call chansend(job1, "ls\n")
:call chansend(job1, "invalid-command\n")
:call chansend(job1, "exit\n")
```

A job may be killed at any time with the jobstop() function:

```
:call jobstop(job1)
```

Individual streams can be closed without killing the job, see chanclose().

LSP client/framework LSP vim.lsp

Nvim supports the Language Server Protocol (LSP), which means it acts as a client to LSP servers and includes a Lua framework `vim.lsp` for building enhanced LSP tools.

https://microsoft.github.io/language-server-protocol/

LSP facilitates features like go-to-definition, find references, hover, completion, rename, format, refactor, etc., using semantic whole-project analysis (unlike ctags).

QUICKSTART lsp-quickstart

Nvim provides an LSP client, but the servers are provided by third parties. Follow these steps to get LSP features:

1. Install language servers using your package manager or by following the upstream installation instructions. You can find language servers here: https://microsoft.github.io/language-server-protocol/implementors/servers/
2. Define a new config lsp-new-config (or install https://github.com/neovim/nvim-lspconfig). Example:

```
vim.lsp.config['emmylua_ls'] = {
  -- Command and arguments to start the server.
  cmd = { 'emmylua_ls' },
  -- Filetypes to automatically attach to.
  filetypes = { 'lua' },
  -- Sets the workspace "root" to the directory where any of these files is found.
  -- Files sharing a root will reuse the LSP client/connection.
  -- Nested lists indicate equal priority, see |vim.lsp.Config|.
  root_markers = { { '.emmyrc.json', '.luarc.json' }, '.git' },
  -- Server-specific settings. https://github.com/EmmyLuaLs/emmylua-analyzer-rust/blob/main/docs/config/emmyrc_json_EN.md
  settings = {
    runtime = {
      version = 'LuaJIT',
    }
  }
}
```

3. Use vim.lsp.enable() to enable the config. See lsp-activate. Example:

```
vim.lsp.enable('emmylua_ls')
```

4. Open a code file matching one of the `filetypes` specified in the config. **Note:** Depending on the LSP server, you may need to ensure your project has a lsp-root_markers file so the workspace can be recognized.
5. Check that LSP is active ("attached") for the buffer:

```
:checkhealth vim.lsp
```

6. **Note:** some LSP features are disabled by default, you can enable them manually:
   - lsp-codelens
   - lsp-linked_editing_range
   - lsp-inlay_hint
   - lsp-inline_completion
7. (Optional) Configure keymaps and autocommands to use LSP features. lsp-attach

DEFAULTS lsp-defaults

When LSP activates, by default it enables various LSP features and sets options and keymaps, listed below, if (1) the language server supports the functionality and (2) the options are empty or were set by the builtin runtime (ftplugin) files. The options are not restored when the LSP client is stopped or detached.

GLOBAL DEFAULTS gra gri grn grr grt grx i_CTRL-S

These GLOBAL keymaps are created unconditionally when Nvim starts:

- "gra" (Normal and Visual mode) is mapped to vim.lsp.buf.code_action()
- "gri" is mapped to vim.lsp.buf.implementation()
- "grn" is mapped to vim.lsp.buf.rename()
- "grr" is mapped to vim.lsp.buf.references()
- "grt" is mapped to vim.lsp.buf.type_definition()
- "grx" is mapped to vim.lsp.codelens.run()
- "gO" is mapped to vim.lsp.buf.document_symbol()
- CTRL-S (Insert mode) is mapped to vim.lsp.buf.signature_help()
- v_an and v_in fall back to LSP vim.lsp.buf.selection_range() if treesitter is not active.
- gx handles `textDocument/documentLink`. Example: with gopls, invoking gx on "os" in this Go code will open documentation externally:

```
package nvim
import (
   "os"
)
```

These LSP features are enabled by default:

- Diagnostics lsp-diagnostic. See vim.diagnostic.config() to customize.
- `workspace/didChangeWatchedFiles` (except on Linux). If you see poor performance in big workspaces, run `:checkhealth vim.lsp` and look for "file watching". Try disabling file-watching:

```
local capabilities = vim.lsp.protocol.make_client_capabilities()
if capabilities.workspace then
  capabilities.workspace.didChangeWatchedFiles = nil
end
vim.lsp.config('*', {
  capabilities = capabilities,
})
```

BUFFER-LOCAL DEFAULTS

- 'omnifunc' is set to vim.lsp.omnifunc(), use i_CTRL-X_CTRL-O to trigger completion.
- 'tagfunc' is set to vim.lsp.tagfunc(). This enables features like go-to-definition, :tjump, and keymaps like CTRL-], CTRL-W_], CTRL-W_} to utilize the language server.
- 'formatexpr' is set to vim.lsp.formatexpr(), so you can format lines via gq if the language server supports it.
  - To opt out of this use gw instead of gq, or clear 'formatexpr' on LspAttach.
- K is mapped to vim.lsp.buf.hover() unless 'keywordprg' is customized or a custom keymap for K exists.
- Document colors are enabled for highlighting color references in a document.
  - To opt out call `vim.lsp.document_color.enable(false, { bufnr = ev.buf })` on LspAttach.

DISABLING DEFAULTS **lsp-defaults-disable**

You can remove GLOBAL keymaps at any time using vim.keymap.del() or :unmap. See also gr-default.

To remove or override BUFFER-LOCAL defaults, define a LspAttach handler:

```
vim.api.nvim_create_autocmd('LspAttach', {
  callback = function(ev)
    -- Unset 'formatexpr'
    vim.bo[ev.buf].formatexpr = nil
    -- Unset 'omnifunc'
    vim.bo[ev.buf].omnifunc = nil
    -- Unmap K
    vim.keymap.del('n', 'K', { buf = ev.buf })
    -- Disable document colors
    vim.lsp.document_color.enable(false, { bufnr = ev.buf })
  end,
})
```

COMMANDS **:lsp lsp-commands E5800**

E5801

The following commands operate on lsp-configs:

:lsp-enable E5802 E5803 :lsp enable [config_name] Activates LSP for current and future buffers. See vim.lsp.enable().

:lsp-disable E5804 :lsp disable [config_name] Disables LSP (and stops if running) for current and future buffers. See vim.lsp.enable().

E5805 E5806 The following commands operate on running clients:

:lsp-restart
:lsp restart [client_name] Restarts LSP clients and servers. If no client names are given, all active clients attached to the current buffer are restarted.

:lsp-stop
:lsp stop [client_name] Stops LSP clients and servers. If no client names are given, all active clients attached to the current buffer are stopped. Use Client:stop() for non-interactive use.

CONFIG **lsp-config**

You can configure LSP behavior statically via vim.lsp.config(), and dynamically via lsp-attach or Client:on_attach().

Use vim.lsp.config() to define or modify LSP configurations, and vim.lsp.enable() to auto-activate them. This is basically a wrapper around vim.lsp.start() which allows you to share and merge configs (provided by Nvim, plugins, and your local config).

NEW CONFIG **lsp-new-config**

To create a new config you can either use `vim.lsp.config()` or create a `lsp/<config-name>.lua` file.

EXAMPLE: DEFINE A CONFIG AS CODE

1. Run `:lua vim.lsp.config('foo', {cmd={'true'}})`
2. Run `:lua vim.lsp.enable('foo')`
3. Run `:checkhealth vim.lsp`, check "Enabled Configurations". 😎

EXAMPLE: DEFINE A CONFIG AS A FILE

1. Create a file `lsp/foo.lua` somewhere on your 'runtimepath'.

```
:exe 'edit' stdpath('config') .. '/lsp/foo.lua'
```

2. Add this code to the file (or copy an example from https://github.com/neovim/nvim-lspconfig):

```
return {
  cmd = { 'true' },
}
```

3. Save the file (with ++p to ensure its parent directory is created).

```
:write ++p
```

4. Enable the config.

```
:lua vim.lsp.enable('foo')
```

5. Run `:checkhealth vim.lsp`, check "Enabled Configurations". 🌈

HOW CONFIGS ACTIVATE

lsp-activate

Call vim.lsp.enable() to ENABLE a config: it will automatically ACTIVATE (attach a client) whenever you open a buffer, if:

- 'filetype' matches one of the config `filetypes`
- 'buftype' is empty or "help"
- `root_markers` or `root_dir()` resolved a workspace root, or the config has `workspace_required=false`

Note: The initial call to vim.lsp.enable() (or :lsp-enable) also actively checks and activates EXISTING buffers, if any.

HOW CONFIGS ARE MERGED

lsp-config-merge

When an LSP client starts, it resolves its configuration by merging the following sources (merge semantics defined by vim.tbl_deep_extend() with "force" behavior), in order of increasing priority:

1. Configuration defined for the `'*'` name.
2. The merged configuration of all `lsp/<config>.lua` files in 'runtimepath' for the config named `<config>`.
3. The merged configuration of all `after/lsp/<config>.lua` files in 'runtimepath'.
   - This behavior of the "after/" directory is a standard Vim feature after-directory which allows you to override `lsp/*.lua` configs provided by plugins (such as nvim-lspconfig).
4. Configurations defined anywhere else.

Example: given the following configs...

```
-- Defined in init.lua
vim.lsp.config('*', {
  capabilities = {
    textDocument = {
      semanticTokens = {
        multilineTokenSupport = true,
      }
    }
  },
  root_markers = { '.git' },
})
-- Defined in <rtp>/lsp/clangd.lua
return {
  cmd = { 'clangd' },
  root_markers = { '.clangd', 'compile_commands.json' },
  filetypes = { 'c', 'cpp' },
}
-- Defined in init.lua
vim.lsp.config('clangd', {
  filetypes = { 'c' },
})
```

...the merged result is:

```
{
  -- From the clangd configuration in <rtp>/lsp/clangd.lua
  cmd = { 'clangd' },

  -- From the clangd configuration in <rtp>/lsp/clangd.lua
  -- Overrides the "*" configuration in init.lua
  root_markers = { '.clangd', 'compile_commands.json' },

  -- From the clangd configuration in init.lua
  -- Overrides the clangd configuration in <rtp>/lsp/clangd.lua
  filetypes = { 'c' },

  -- From the "*" configuration in init.lua
  capabilities = {
    textDocument = {
      semanticTokens = {
        multilineTokenSupport = true,
      }
    }
  }
}
```

CONFIGURE ON ATTACH

lsp-attach

To use LSP features beyond those provided by Nvim (see lsp-buf), you can set keymaps and options on Client:on_attach() or LspAttach. Not all language servers provide the same capabilities; check `supports_method()` in your LspAttach handler. lsp-lint lsp-format Example: Enable auto-completion and auto-formatting ("linting"):

```
vim.api.nvim_create_autocmd('LspAttach', {
  group = vim.api.nvim_create_augroup('my.lsp'),
  callback = function(ev)
    local client = assert(vim.lsp.get_client_by_id(ev.data.client_id))
    if client:supports_method('textDocument/implementation') then
      -- Create a keymap for vim.lsp.buf.implementation ...
    end

    -- Enable auto-completion. Note: Use CTRL-Y to select an item. |complete_CTRL-Y|
    if client:supports_method('textDocument/completion') then
      -- Optional: trigger autocompletion on EVERY keypress. May be slow!
      -- local chars = {}; for i = 32, 126 do table.insert(chars, string.char(i)) end
      -- client.server_capabilities.completionProvider.triggerCharacters = chars

      vim.lsp.completion.enable(true, client.id, ev.buf, {autotrigger = true})
    end

    -- Auto-format ("lint") on save.
    -- Usually not needed if server supports "textDocument/willSaveWaitUntil".
    if not client:supports_method('textDocument/willSaveWaitUntil')
        and client:supports_method('textDocument/formatting') then
      vim.api.nvim_create_autocmd('BufWritePre', {
        group = vim.api.nvim_create_augroup('my.lsp', {clear=false}),
        buffer = ev.buf,
        callback = function()
          vim.lsp.buf.format({ bufnr = ev.buf, id = client.id, timeout_ms = 1000 })
        end,
      })
    end
  end,
})
```

lsp-buf-working-dir

The LSP server decides the "workspace root" (lsp-root_dir()) of a buffer, but the Nvim current-directory does not follow it, so commands like :make, :grep and :terminal do not run from the workspace root. You can use :bcd to sync each buffer's directory to its LSP workspace root:

```
vim.api.nvim_create_autocmd('LspAttach', {
  group = vim.api.nvim_create_augroup('my.lsp.bufdir', {}),
  callback = function(ev)
    local client = assert(vim.lsp.get_client_by_id(ev.data.client_id))
    if client.root_dir then
      vim.cmd.bcd(client.root_dir)
    end
  end,
})
```

To see the capabilities for a given server, try this in a LSP-enabled buffer:

```
:lua =vim.lsp.get_clients()[1].server_capabilities
```

FAQ

lsp-faq

- Q: How to force-reload LSP?
- A: Use `:lsp restart`. You can also stop all clients, then reload the buffer:

```
:lsp stop
:edit
```

- Q: Why isn't completion working?
- A: In the buffer where you want to use LSP, check that 'omnifunc' is set to "v:lua.vim.lsp.omnifunc": `:verbose set omnifunc?`
  - Some other plugin may be overriding the option. To avoid that you could set the option in an after-directory ftplugin, e.g. "after/ftplugin/python.vim".
- Q: How do I run a request synchronously (e.g. for formatting on file save)?
- A: Check if the function has an `async` parameter and set the value to false. E.g. code formatting:

```
" Auto-format *.rs (rust) files prior to saving them
" (async = false is the default for format)
autocmd BufWritePre *.rs lua vim.lsp.buf.format({ async = false })
```

- Q: How to avoid my own lsp/ folder being overridden?
- A: Place your configs under "after/lsp/". Files in "after/lsp/" are loaded after those in "nvim/lsp/", so your settings will take precedence over the defaults provided by nvim-lspconfig. See also: after-directory

lsp-vs-treesitter

- Q: How do LSP, Treesitter and Ctags compare?
- A: LSP requires a client and language server. The language server uses semantic analysis to understand code at a project level. This provides language servers with the ability to rename across files, find definitions in external libraries and more.

treesitter is a language parsing library that provides excellent tools for incrementally parsing text and handling errors. This makes it a great fit for editors to understand the contents of the current file for things like syntax highlighting, simple goto-definitions, scope analysis and more.

A ctags-like program can generate a tags file that allows Nvim to jump to definitions, provide simple completions via i_CTRL-X_CTRL-] command. It is not as featureful and doesn't have semantic understanding, but it is fast, lightweight and useful for navigating polyglot projects.

CONCEPTS

API

lsp-api

The lsp-core API provides core functions for creating and managing clients, You can even create creating custom (in-process) lsp-servers. The lsp-buf functions operate LSP clients attached to the current buffer.

METHOD

lsp-method

Requests and notifications defined by the LSP specification are referred to as "LSP methods". These are handled by lsp-handler functions.

HANDLER

lsp-handler

LSP handlers are functions that handle lsp-responses to requests made by Nvim to the server. (Notifications, as opposed to requests, are fire-and-forget: there is no response, so they can't be handled. lsp-notification)

Each handler has the following signature:

```
vim.lsp.ResponseHandler:
    fun(err, result, ctx)

vim.lsp.NotificationHandler:
    fun(err, params, ctx)

vim.lsp.RequestHandler:
    fun(err, params, ctx): Result?, lsp.ResponseError?
```

Each response handler has this signature:

```
function(err, result, ctx)
```

Parameters:

- `{err}` (`table|nil`) Error info dict, or `nil` if the request completed.
- `{result}` (`Result|Params|nil`) `result` key of the lsp-response or `nil` if the request failed.
- `{ctx}` (`table`) Table of calling state associated with the handler, with these keys:
  - `{method}` (`string`) lsp-method name.
  - `{client_id}` (`number`) vim.lsp.Client identifier.
  - `{bufnr}` (`Buffer`) Buffer handle.
  - `{params}` (`table|nil`) Request parameters table.
  - `{version}` (`number`) Document version at time of request. Handlers can compare this to the current document version to check if the response is "stale". See also b:changedtick.

Return (multiple):

- (`Result?`) `result` on success, or `nil` on error.
- (`lsp.ResponseError?`) `error` on failure, or `nil` on success. RPC error shape:

  ```
  { code, message, data? }
  ```

  You can use vim.lsp.rpc.rpc_response_error() to create this object.

lsp-response and lsp-notification handlers do not have return values.

lsp-handler-resolution

Handlers can be set by (in increasing priority):

vim.lsp.handlers

- Directly calling a LSP method via Client:request(). This is the only way to "override" the default client-to-server request handling (by side-stepping `vim.lsp.buf` and related interfaces).

  ```
  local client = assert(vim.lsp.get_clients()[1])
  client:request('textDocument/definition')
  ```

- Setting a field in `vim.lsp.handlers`. This global table contains the default mappings of lsp-method names to handlers. (**Note:** only for server-to-client requests/notifications, not client-to-server.) Example:

  ```
  vim.lsp.handlers['textDocument/publishDiagnostics'] = my_custom_diagnostics_handler
  ```

- Passing a `{handlers}` parameter to vim.lsp.start(). This sets the default lsp-handler for a specific server. (**Note:** only for server-to-client requests/notifications, not client-to-server.) Example:

  ```
  vim.lsp.start {
    ..., -- Other configuration omitted.
    handlers = {
      ['textDocument/publishDiagnostics'] = my_custom_diagnostics_handler
    },
  }
  ```

- Passing a `{handler}` parameter to vim.lsp.buf_request_all(). This sets the lsp-handler ONLY for the given request(s). Example:

```
vim.lsp.buf_request_all(
  0,
  'textDocument/publishDiagnostics',
  my_request_params,
  my_handler
)
```

vim.lsp.log_levels

Log levels are defined in vim.log.levels

PROTOCOL vim.lsp.protocol

Module `vim.lsp.protocol` defines constants dictated by the LSP specification, and helper functions for creating protocol-related objects.
https://github.com/microsoft/language-server-protocol/raw/gh-pages/_specifications/specification-3-14.md

For example `vim.lsp.protocol.ErrorCodes` allows reverse lookup by number or name:

```
vim.lsp.protocol.TextDocumentSyncKind.Full == 1
vim.lsp.protocol.TextDocumentSyncKind[1] == "Full"
```

lsp-request

LSP request shape:

```
{ id: integer|string, method: string, params?: Params }
```

https://microsoft.github.io/language-server-protocol/specifications/specification-current/#requestMessage

lsp-response

LSP response shape:

```
{ id: integer|string|nil, result: Result, error: nil }       (on success)
{ id: integer|string|nil, result: nil, error: ResponseError }  (on error)
```

https://microsoft.github.io/language-server-protocol/specifications/specification-current/#responseMessage

lsp-notification

LSP notification shape:

```
{ method: string, params?: Params }
```

https://microsoft.github.io/language-server-protocol/specifications/specification-current/#notificationMessage

IN-PROCESS LSP SERVER lsp-server

For maximum flexibility you may want to create your own server. This allows you to hook into Nvim LSP features automatically. The "server" can be an in-process Lua function instead of an external process. For example see `runtime/lua/vim/pack/_lsp.lua` which provides features like "docs hover" in the vim.pack "Confirm Updates" UI.

This is possible because vim.lsp.start() accepts `cmd` as a Lua function. The function must accept `vim.lsp.rpc.Dispatchers` and return a table in the form of vim.lsp.rpc.Client. Use `dispatchers.on_exit()` to signal that the server has stopped.

Example (select and run the code with `:lua`, then check `:che vim.lsp` and look for "my-server"):

```
--- @return vim.lsp.rpc.Client
local function cmd_fn(dispatchers)
  local closing = false
  local request_id = 0

  local srv = {} --[[@type vim.lsp.rpc.Client]]
  function srv.request(method, params, callback)
    if method == 'initialize' then
      callback(nil, {
        capabilities = {
          hoverProvider = true,
        },
      })
    elseif method == 'shutdown' then
      callback(nil, nil)
    end
    request_id = request_id + 1
    return true, request_id
  end
  function srv.notify(method, params)
    if method == 'exit' then
      dispatchers.on_exit(0, 15)
    end
  end
  function srv.is_closing()
    return closing
  end
  function srv.terminate()
    closing = true
  end

  return srv
end

-- Define a config for the server, then enable it...
vim.lsp.config['my-server'] = {
  cmd = cmd_fn,
  filetypes = { 'lua' },
  root_markers = { '.git' },
}
vim.lsp.enable('my-server')

-- ...or call start() directly.
vim.lsp.start(
  { cmd = cmd_fn, name = 'my-server', root_dir = vim.uv.cwd() },
  { attach = true })
```

LSP HIGHLIGHT

lsp-highlight

Reference Highlights:

Highlight groups that are meant to be used by vim.lsp.buf.document_highlight().

You can see more about the differences in types here: https://microsoft.github.io/language-server-protocol/specification#textDocument_documentHighlight

hl-LspReferenceText
hl-LspReferenceRead
hl-LspReferenceWrite
hl-LspReferenceTarget
hl-LspInlayHint

LspReferenceText used for highlighting "text" references
LspReferenceRead used for highlighting "read" references
LspReferenceWrite used for highlighting "write" references
LspReferenceTarget used for highlighting reference targets (e.g. in a hover range)
LspInlayHint used for highlighting inlay hints

lsp-highlight-codelens

Highlight groups related to lsp-codelens functionality.

hl-LspCodeLens

LspCodeLens Used to color the virtual text of the codelens. See nvim_buf_set_extmark().

hl-LspCodeLensSeparator

LspCodeLensSeparator
Used to color the separator between two or more code lenses.

lsp-highlight-signature

Highlight groups related to vim.lsp.handlers.signature_help().

hl-LspSignatureActiveParameter

LspSignatureActiveParameter Used to highlight the active parameter in the signature help. See vim.lsp.handlers.signature_help().

LSP SEMANTIC HIGHLIGHTS

lsp-semantic-highlight

When available, the LSP client highlights code using lsp-semantic_tokens, which are another way that LSP servers can provide information about source code. Note that this is in addition to treesitter syntax highlighting; semantic highlighting does not replace syntax highlighting.

The server will typically provide one token per identifier in the source code. The token will have a `type` such as "function" or "variable", and 0 or more `modifiers` such as "readonly" or "deprecated." The standard types and modifiers are described here:

https://microsoft.github.io/language-server-protocol/specification/#textDocument_semanticTokens LSP servers may also use off-spec types and modifiers.

The LSP client adds one or more highlights for each token. The highlight groups are derived from the token's type and modifiers:

- `@lsp.type.<type>.<ft>` for the type
- `@lsp.mod.<mod>.<ft>` for each modifier
- `@lsp.typemod.<type>.<mod>.<ft>` for each modifier Use :Inspect to view the highlights for a specific token. Use :hi or nvim_set_hl() to change the appearance of semantic highlights:

```
hi @lsp.type.function guifg=Yellow        " function names are yellow
hi @lsp.type.variable.lua guifg=Green     " variables in lua are green
hi @lsp.mod.deprecated gui=strikethrough  " deprecated is crossed out
hi @lsp.typemod.function.async guifg=Blue " async functions are blue
```

The value vim.hl.priorities`.semantic_tokens` is the priority of the `@lsp.type.*` highlights. The `@lsp.mod.*` and `@lsp.typemod.*` highlights have priorities one and two higher, respectively.

You can disable semantic highlights by clearing the highlight groups:

```
-- Hide semantic highlights for functions
vim.api.nvim_set_hl(0, '@lsp.type.function', {})

-- Hide all semantic highlights
for _, group in ipairs(vim.fn.getcompletion("@lsp", "highlight")) do
  vim.api.nvim_set_hl(0, group, {})
end
```

You probably want these inside a ColorScheme autocommand.

Use LspTokenUpdate and vim.lsp.semantic_tokens.highlight_token() for more complex highlighting.

The following is a list of standard captures used in queries for Nvim, highlighted according to the current colorscheme (use :Inspect on one to see the exact definition):

@lsp.type.class Identifiers that declare or reference a class type @lsp.type.comment Tokens that represent a comment @lsp.type.decorator Identifiers that declare or reference decorators and annotations @lsp.type.enum Identifiers that declare or reference an enumeration type @lsp.type.enumMember Identifiers that declare or reference an enumeration property, constant, or member @lsp.type.event Identifiers that declare an event property @lsp.type.function Identifiers that declare a function @lsp.type.interface Identifiers that declare or reference an interface type @lsp.type.keyword Tokens that represent a language keyword @lsp.type.macro Identifiers that declare a macro @lsp.type.method Identifiers that declare a member function or method @lsp.type.modifier Tokens that represent a modifier @lsp.type.namespace Identifiers that declare or reference a namespace, module, or package @lsp.type.number Tokens that represent a number literal @lsp.type.operator Tokens that represent an operator @lsp.type.parameter Identifiers that declare or reference a function or method parameters @lsp.type.property Identifiers that declare or reference a member property, member field, or member variable @lsp.type.regexp Tokens that represent a regular expression literal @lsp.type.string Tokens that represent a string literal @lsp.type.struct Identifiers that declare or reference a struct type @lsp.type.type Identifiers that declare or reference a type that is not covered above @lsp.type.typeParameter Identifiers that declare or reference a type parameter @lsp.type.variable Identifiers that declare or reference a local or global variable

@lsp.mod.abstract Types and member functions that are abstract @lsp.mod.async Functions that are marked async @lsp.mod.declaration Declarations of symbols @lsp.mod.defaultLibrary Symbols that are part of the standard library @lsp.mod.definition Definitions of symbols, for example, in header files @lsp.mod.deprecated Symbols that should no longer be used @lsp.mod.documentation Occurrences of symbols in documentation @lsp.mod.modification Variable references where the variable is assigned to @lsp.mod.readonly Readonly variables and member fields (constants) @lsp.mod.static Class members (static members)

EVENTS

lsp-events

LspAttach LspAttach

After an LSP client performs "initialize" and attaches to a buffer. The autocmd-pattern is the buffer name. The event-data (type: `vim.event.lspattach.data`) has the client ID.

Example:

```
vim.api.nvim_create_autocmd('LspAttach', {
  callback = function(ev)
    local client = vim.lsp.get_client_by_id(ev.data.client_id)
    -- ...
  end
})
```

Note: If the LSP server performs dynamic registration, capabilities may be registered any time _after_ LspAttach. In that case you may want to handle the "registerCapability" event.

Example:

```
vim.lsp.handlers['client/registerCapability'] = (function(overridden)
  return function(err, res, ctx)
    local result = overridden(err, res, ctx)
    local client = vim.lsp.get_client_by_id(ctx.client_id)
    if not client then
      return
    end
    for bufnr, _ in pairs(client.attached_buffers) do
      -- Call your custom on_attach logic...
      -- my_on_attach(client, bufnr)
    end
    return result
  end
end)(vim.lsp.handlers['client/registerCapability'])
```

LspDetach LspDetach

Just before an LSP client detaches from a buffer. The autocmd-pattern is the buffer name. The event-data (type: `vim.event.lspdetach.data`) has the client ID.

Example:

```
vim.api.nvim_create_autocmd('LspDetach', {
  callback = function(ev)
    -- Get the detaching client
    local client = vim.lsp.get_client_by_id(ev.data.client_id)

    -- Remove the autocommand to format the buffer on save, if it exists
    if client:supports_method('textDocument/formatting') then
      vim.api.nvim_clear_autocmds({
        event = 'BufWritePre',
        buffer = ev.buf,
      })
    end
  end,
})
```

LspNotify LspNotify

This event is triggered after each successful notification sent to an LSP server.

The event-data (type: `vim.event.lspnotify.data`) has the client_id, LSP method, and parameters.

Example:

```
vim.api.nvim_create_autocmd('LspNotify', {
  callback = function(ev)
    local bufnr = ev.buf
    local client_id = ev.data.client_id
    local method = ev.data.method
    local params = ev.data.params

    -- do something with the notification
    if method == 'textDocument/...' then
      update_buffer(bufnr)
    end
  end,
})
```

LspProgress LspProgress

Upon receipt of a progress notification from the server. Notifications can be polled from a `progress` ring buffer of a vim.lsp.Client or use vim.lsp.status() to get an aggregate message.

If the server sends a "work done progress", the `pattern` is set to `kind` (one of `begin`, `report` or `end`).

The event-data (type: `vim.event.lspprogress.data`) has `client_id` and `params` properties, where `params` is the request params sent by the server (see `lsp.ProgressParams`).

Examples:

Redraw the statusline whenever an LSP progress message arrives:

```
autocmd LspProgress * redrawstatus
```

Emit a progress-message on LSP progress events:

```
vim.api.nvim_create_autocmd('LspProgress', { buffer = buf, callback = function(ev)
    local value = ev.data.params.value
    vim.api.nvim_echo({ { value.message or 'done' } }, false, {
      id = 'lsp.' .. ev.data.params.token,
      kind = 'progress',
      source = 'vim.lsp',
      title = value.title,
      status = value.kind ~= 'end' and 'running' or 'success',
      percent = value.percentage,
    })
  end,
})
```

See also:

- https://github.com/MicrosoftDocs/terminal/blob/main/TerminalDocs/tutorials/progress-bar-sequences.md

LspRequest LspRequest

For each request sent to an LSP server, this event is triggered for every change to the request's status. The status can be one of `pending`, `complete`, or `cancel` and is sent as the `{type}` on the "data" table passed to the callback function.

It triggers when the initial request is sent (`{type} == pending`) and when the LSP server responds (`{type} == complete`). If a cancellation is requested using `client.cancel_request(request_id)`, then this event will trigger with `{type} == cancel`.

The event-data (type: `vim.event.lsprequest.data`) has the client ID, request ID, and request (described at vim.lsp.Client, `{requests}` field). If the request type is `complete`, the request will be deleted from the client's pending requests table after processing the event handlers.

Example:

```
vim.api.nvim_create_autocmd('LspRequest', {
  callback = function(ev)
    local bufnr = ev.buf
    local client_id = ev.data.client_id
    local request_id = ev.data.request_id
    local request = ev.data.request
    if request.type == 'pending' then
      -- do something with pending requests
      track_pending(client_id, bufnr, request_id, request)
    elseif request.type == 'cancel' then
      -- do something with pending cancel requests
      track_canceling(client_id, bufnr, request_id, request)
    elseif request.type == 'complete' then
      -- do something with finished requests. this pending
      -- request entry is about to be removed since it is complete
      track_finish(client_id, bufnr, request_id, request)
    end
  end,
})
```

LspTokenUpdate LspTokenUpdate

When a visible semantic token is sent or updated by the LSP server, or when an existing token becomes visible for the first time. The autocmd-pattern is the buffer name. The event-data (type: `vim.event.lsptokenupdate.data`) has the client ID and token (see vim.lsp.semantic_tokens.get_at_pos()).

Example:

```
vim.api.nvim_create_autocmd('LspTokenUpdate', {
  callback = function(ev)
    local token = ev.data.token
    if token.type == 'variable' and not token.modifiers.readonly then
      vim.lsp.semantic_tokens.highlight_token(
        token, ev.buf, ev.data.client_id, 'MyMutableVariableHighlight'
      )
    end
  end,
})
```

Note: doing anything other than calling vim.lsp.semantic_tokens.highlight_token() is considered experimental.

Lua module: vim.lsp lsp-core

vim.lsp.Config Extends: vim.lsp.ClientConfig

Fields:

- `{cmd}?` (`string[]|fun(dispatchers: vim.lsp.rpc.Dispatchers, config: vim.lsp.ClientConfig): vim.lsp.rpc.Client`) See `cmd` in vim.lsp.ClientConfig. See also `reuse_client` to dynamically decide (per-buffer) when `cmd` should be re-invoked.
- `{filetypes}?` (`string[]`) Filetypes the client will attach to, or `nil` for ALL filetypes. To match files by name, pattern, or contents, you can define a custom filetype using vim.filetype.add():

  ```
  vim.filetype.add({
    filename = {
      ['my_filename'] = 'my_filetype1',
    },
    pattern = {
      ['.*/etc/my_file_pattern/.*'] = 'my_filetype2',
    },
  })
  vim.lsp.config('…', {
    filetypes = { 'my_filetype1', 'my_filetype2' },
  })
  ```

- `{reuse_client}?` (`fun(client: vim.lsp.Client, config: vim.lsp.ClientConfig, bufnr: integer): boolean`) Predicate which decides if a client should be re-used. Used on all running clients. The default implementation re-uses a client if name and root_dir matches.
- `{root_dir}?` (`string|fun(bufnr: integer, on_dir:fun(root_dir?:string))`) lsp-root_dir() Decides the workspace root: the directory where the LSP server will base its workspaceFolders, rootUri, and rootPath on initialization. The function form must call the `on_dir` callback to provide the root dir, or LSP will not be activated for the buffer. Thus a `root_dir()` function can dynamically decide per-buffer whether to activate (or skip) LSP. See example at vim.lsp.enable().
- `{root_markers}?` (`((string|string[])[])`) lsp-root_markers
  Filename(s) ("".git/"", ""package.json"", …) used to decide the workspace root. Unused if `root_dir` is defined. The list order decides priority. To indicate "equal priority", specify names in a nested list `{ { 'a.txt', 'b.lua' }, ... }`.
  - For each item, Nvim will search upwards (from the buffer file) for that marker, or list of markers; search stops at the first directory containing that marker, and the directory is used as the root dir (workspace folder).
  - Example: Find the first ancestor directory containing file or directory "stylua.toml"; if not found, find the first ancestor containing ".git":

```
root_markers = { 'stylua.toml', '.git' }
```

- Example: Find the first ancestor directory containing EITHER "stylua.toml" or ".luarc.json"; if not found, find the first ancestor containing ".git":

```
root_markers = { { 'stylua.toml', '.luarc.json' }, '.git' }
```

buf_attach_client({bufnr}, {client_id}) vim.lsp.buf_attach_client()
Implements the `textDocument/did…` notifications required to track a buffer for any language server.

Without calling this, the server won't be notified of changes to a buffer.

Parameters:
- {bufnr} (integer) Buffer handle, or 0 for current
- {client_id} (integer) Client id

Return:
(boolean) success `true` if client was attached successfully; `false` otherwise

buf_detach_client({bufnr}, {client_id}) vim.lsp.buf_detach_client()
Detaches client from the specified buffer. **Note:** While the server is notified that the text document (buffer) was closed, it is still able to send notifications should it ignore this notification.

Parameters:
- {bufnr} (integer) Buffer handle, or 0 for current
- {client_id} (integer) Client id

buf_is_attached({bufnr}, {client_id}) vim.lsp.buf_is_attached()
Checks if a buffer is attached for a particular client.

Parameters:
- {bufnr} (integer) Buffer handle, or 0 for current
- {client_id} (integer) the client id

buf_notify({bufnr}, {method}, {params}) vim.lsp.buf_notify()
Send a notification to a server

Attributes:
Since: 0.5.0

Parameters:
- {bufnr} (integer?) The number of the buffer
- {method} (string) Name of the request method
- {params} (any) Arguments to send to the server

Return:
(boolean) success true if any client returns true; false otherwise

vim.lsp.buf_request_all()
buf_request_all({bufnr}, {method}, {params}, {handler}) Sends an async request for all active clients attached to the buffer and executes the `handler` callback with the combined result.

Attributes:
Since: 0.5.0

Parameters:
- {bufnr} (integer) Buffer handle, or 0 for current.
- {method} (string) LSP method name
- {params} (table|(fun(client: vim.lsp.Client, bufnr: integer): table?)?) Parameters to send to the server. Can also be passed as a function that returns the params table for cases where parameters are specific to the client.
- {handler} (function) Handler called after all requests are completed. Server results are passed as a `client_id:result` map.

Return:
(function) cancel Function that cancels all requests.

vim.lsp.buf_request_sync()
buf_request_sync({bufnr}, {method}, {params}, {timeout_ms}) Sends a request to all server and waits for the response of all of them.

Calls vim.lsp.buf_request_all() but blocks Nvim while awaiting the result. Parameters are the same as vim.lsp.buf_request_all() but the result is different. Waits a maximum of {timeout_ms}.

Attributes:
Since: 0.5.0

Parameters:
- {bufnr} (integer) Buffer handle, or 0 for current.
- {method} (string) LSP method name
- {params} (table|(fun(client: vim.lsp.Client, bufnr: integer): table?)?) Parameters to send to the server. Can also be passed as a function that returns the params table for cases where parameters are specific to the client.
- {timeout_ms} (integer?, default: 1000) Maximum time in milliseconds to wait for a result.

Return (multiple):
(table<integer, {error: lsp.ResponseError?, result: any}>?) result Map of client_id:request_result. (string?) err On timeout, cancel, or error, `err` is a string describing the failure reason, and `result` is nil.

commands vim.lsp.commands
Map of client-defined handlers implementing custom (off-spec) commands which a server may invoke. Each key is a unique command name; each value is a function which is called when an LSP action (code action, code lenses, …) requests it by name.

If an LSP response requests a command not defined client-side, Nvim will forward it to the server as `workspace/executeCommand`.

- Argument 1 is the `Command`:

```
Command
title: String
command: String
arguments?: any[]
```

- Argument 2 is the lsp-handler `ctx`.

Example:

```
vim.lsp.commands['java.action.generateToStringPrompt'] = function(_, ctx)
  require("jdtls.async").run(function()
    local _, result = request(ctx.bufnr, 'java/checkToStringStatus', ctx.params)
    local fields = ui.pick_many(result.fields, 'Include item in toString?', function(x)
      return string.format('%s: %s', x.name, x.type)
    end)
    local _, edit = request(ctx.bufnr, 'java/generateToString', { context = ctx.params; fields = fields; })
    vim.lsp.util.apply_workspace_edit(edit, offset_encoding)
  end)
end
```

config({name}, {cfg}) vim.lsp.config()
Sets the default configuration for an LSP client (or all clients if the special name "*" is used).

Can also be accessed by table-indexing (`vim.lsp.config[…]`) to get the resolved config, or redefine the config (instead of "merging" with the config chain).

Examples:
- Add root markers for ALL clients:

```
vim.lsp.config('*', {
  root_markers = { '.git', '.hg' },
})
```

- Add capabilities to ALL clients:

```
  vim.lsp.config('*', {
  capabilities = {
    textDocument = {
      semanticTokens = {
        multilineTokenSupport = true,
      }
    }
  }
})
```

- Add root markers and capabilities for "clangd":

```
  vim.lsp.config('clangd', {
  root_markers = { '.clang-format', 'compile_commands.json' },
  capabilities = {
    textDocument = {
      completion = {
        completionItem = {
          snippetSupport = true,
        }
      }
    }
  }
})
```

- (Re-)define the "clangd" configuration (overrides the resolved chain):

```
  vim.lsp.config.clangd = {
  cmd = {
    'clangd',
    '--clang-tidy',
    '--background-index',
    '--offset-encoding=utf-8',
  },
  root_markers = { '.clangd', 'compile_commands.json' },
  filetypes = { 'c', 'cpp' },
}
```

- Get the resolved configuration for "emmylua_ls":

```
local cfg = vim.lsp.config.emmylua_ls
```

Attributes:
Since: 0.11.0

Parameters:
- `{name}` (`string`)
- `{cfg}` (`vim.lsp.Config`)

enable({name}, {enable}) vim.lsp.enable()
Enables a lsp-config: automatically attaches the client to any buffer based on the config `filetypes`, `root_markers`, and `root_dir`. See lsp-activate for details.

To disable, pass `enable=false`: Stops related clients and servers (force-stops servers after a timeout, unless `exit_timeout=false`).

Raises an error under the following conditions:
- {name} is not a valid LSP config name (for example, `'*'`).
- {name} corresponds to an LSP config file which raises an error.

If an error is raised when multiple names are provided, this function will have no side-effects; it will not enable/disable any configs, including ones which contain no errors.

Examples:

```
vim.lsp.enable('clangd')
vim.lsp.enable({'emmylua_ls', 'pyright'})
```

Example: To dynamically decide whether LSP is activated, define a lsp-root_dir() function which calls `on_dir()` only when you want that config to activate:

```
vim.lsp.config('emmylua_ls', {
  root_dir = function(bufnr, on_dir)
    if vim.fs.ext(vim.fn.bufname(bufnr)) ~= 'txt' then
      on_dir(vim.fn.getcwd())
    end
  end
})
```

Attributes:
Since: 0.11.0

Parameters:
- {name} (`string|string[]`) Name(s) of client(s) to enable.
- {enable} (`boolean?`) If `true|nil`, enables auto-activation of the given LSP config on current and future buffers. If `false`, disables auto-activation and stops related LSP clients and servers (force-stops servers after `exit_timeout` milliseconds).

foldclose({kind}, {winid}) vim.lsp.foldclose()
Close all {kind} of folds in the window with {winid}.

To automatically fold imports when opening a file, you can use an autocmd:

```
vim.api.nvim_create_autocmd('LspNotify', {
  callback = function(ev)
    if ev.data.method == 'textDocument/didOpen' then
      vim.lsp.foldclose('imports', vim.fn.bufwinid(ev.buf))
    end
  end,
})
```

Attributes:
Since: 0.11.0

Parameters:
- {kind} (`lsp.FoldingRangeKind`) Kind to close, one of "comment", "imports" or "region".
- {winid} (`integer?`) Defaults to the current window.

foldexpr({lnum}) vim.lsp.foldexpr()
Provides an interface between the built-in client and a `foldexpr` function.

To use, set 'foldmethod' to "expr" and set the value of 'foldexpr':

```
vim.o.foldmethod = 'expr'
vim.o.foldexpr = vim.lsp.foldexpr
```

Or use it only when supported by checking for the "textDocument/foldingRange" capability in an LspAttach autocommand. Example:

```
vim.o.foldmethod = 'expr'
-- Default to treesitter folding
vim.o.foldexpr = vim.treesitter.foldexpr
-- Prefer LSP folding if client supports it
vim.api.nvim_create_autocmd('LspAttach', {
  callback = function(ev)
    local client = vim.lsp.get_client_by_id(ev.data.client_id)
    if client:supports_method('textDocument/foldingRange') then
      local win = vim.api.nvim_get_current_win()
      vim.wo[win][0].foldexpr = vim.lsp.foldexpr
    end
  end,
})
```

Parameters:
- {lnum} (`integer`) line number

foldtext({lnum}) vim.lsp.foldtext()
Provides a `foldtext` function that shows the `collapsedText` retrieved, defaults to the first folded line if `collapsedText` is not provided.

The displayed foldtext will be highlighted via treesitter.

Parameters:
- {lnum} (`integer?`) line number (default: current fold start)

formatexpr({opts}) vim.lsp.formatexpr()
Provides an interface between the built-in client and a `formatexpr` function.

Currently only supports a single client. This can be set via `vim.bo[bufnr].formatexpr = vim.lsp.formatexpr`, or with a wrapper to pass options:

```
vim.bo[bufnr].formatexpr = function()
  return vim.lsp.formatexpr({ timeout_ms = 250 })
end
```

Parameters:
- `{opts}` (`table?`) A table with the following fields:
  - `{timeout_ms}?` (`integer`, default: 500ms) The timeout period for the formatting request..

get_client_by_id({client_id}) vim.lsp.get_client_by_id()
Gets a client by id, or nil if the id is invalid or the client was stopped. The returned client may not yet be fully initialized.

Parameters:
- `{client_id}` (`integer`) client id

Return:
(`vim.lsp.Client?`) client rpc object

get_clients({filter}) vim.lsp.get_clients()
Gets active clients.

Attributes:
Since: 0.10.0

Parameters:
- `{filter}` (`table?`) Key-value pairs used to filter the returned clients.
  - `{bufnr}?` (`integer`) Only return clients attached to this buffer
  - `{id}?` (`integer`) Only return clients with the given id
  - `{method}?` (`string`) Only return clients supporting the given method
  - `{name}?` (`string`) Only return clients with the given name

Return:
(`vim.lsp.Client[]`) List of vim.lsp.Client objects

get_configs({filter}) vim.lsp.get_configs()
Gets LSP configs.

See also vim.lsp.get_clients() to get the runtime values of dynamic fields like `root_dir`, which depend on the current buffer/workspace/etc.

WARNING: May eagerly (prematurely!) evaluate config files in 'runtimepath'.

Attributes:
Since: 0.12.0

Parameters:
- `{filter}` (`table?`) Key-value pairs used to filter the returned configs.
  - `{enabled}?` (`boolean`) If true, only return enabled configs. If false, only return configs that aren't enabled.
  - `{filetype}?` (`string`) Only return configs which attach to the given filetype.

Return:
(`vim.lsp.Config[]`) List of vim.lsp.Config objects

is_enabled({name}) vim.lsp.is_enabled()
Checks if the given LSP config is enabled (globally, not per-buffer).

Unlike `vim.lsp.config['…']`, this does not have the side-effect of resolving the config.

Parameters:
- `{name}` (`string`) Config name

Return:
(`boolean`)

omnifunc({findstart}, {base}) vim.lsp.omnifunc()
Implements 'omnifunc' compatible LSP completion.

Parameters:
- `{findstart}` (`integer`) 0 or 1, decides behavior
- `{base}` (`integer`) findstart=0, text to match against

Return:
(`integer|table`) Decided by `{findstart}`:
- findstart=1: column where the completion starts, or -2 or -3
- findstart=0: list of matches (actually just calls complete())

See also:
- complete-functions
- complete-items
- CompleteDone

start({config}, {opts}) vim.lsp.start()
Create a new LSP client and start a language server or reuses an already running client if one is found matching `name` and `root_dir`. Attaches the current buffer to the client.

Example:

```
vim.lsp.start({
   name = 'my-server-name',
   cmd = {'name-of-language-server-executable'},
   root_dir = vim.fs.root(0, {'pyproject.toml', 'setup.py'}),
})
```

See vim.lsp.ClientConfig for all available options. The most important are:

- `name` arbitrary name for the LSP client. Should be unique per language server.
- `cmd` command string[] or function. See also lsp-server.
- `root_dir` path to the project root. By default this is used to decide if an existing client should be re-used. The example above uses vim.fs.root() to detect the root by traversing the file system upwards starting from the current directory until either a `pyproject.toml` or `setup.py` file is found.
- `workspace_folders` list of `{ uri:string, name: string }` tables specifying the project root folders used by the language server. If `nil` the property is derived from `root_dir` for convenience.

Language servers use this information to discover metadata like the dependencies of your project and they tend to index the contents within the project folder.

To ensure a language server is only started for languages it can handle, make sure to call vim.lsp.start() within a FileType autocmd. Either use :au, nvim_create_autocmd() or put the call in a `ftplugin/<filetype_name>.lua` (See ftplugin-name)

Attributes:
Since: 0.8.0

Parameters:

- `{config}` (`vim.lsp.ClientConfig`) Configuration for the server.
- `{opts}` (`table?`) Optional keyword arguments.
  - `{attach}?` (`boolean`, default: true) Whether to attach the client to a buffer. If set to `false`, `reuse_client` and `bufnr` will be ignored.
  - `{bufnr}?` (`integer`) Buffer handle to attach to if starting or re-using a client (0 for current).
  - `{reuse_client}?` (`fun(client: vim.lsp.Client, config: vim.lsp.ClientConfig, bufnr: integer): boolean`) Predicate used to decide if a client should be re-used. Used on all running clients. The default implementation re-uses a client if it has the same name and if the given workspace folders (or root_dir) are all included in the client's workspace folders.
  - `{silent}?` (`boolean`, default: false) Suppress error reporting if the LSP server fails to start.

Return:
(`integer?`) client_id

status() vim.lsp.status()
Consumes the latest progress messages from all clients and formats them as a string. Empty if there are no clients or if no new messages

Return:
(`string`)

tagfunc({`pattern`}, {`flags`}) vim.lsp.tagfunc()
Provides an interface between the built-in client and 'tagfunc'.

When used with normal mode commands (e.g. CTRL-]) this will invoke the "textDocument/definition" LSP method to find the tag under the cursor. Otherwise, uses "workspace/symbol". If no results are returned from any LSP servers, falls back to using built-in tags.

Parameters:

- `{pattern}` (`string`) Pattern used to find a workspace symbol
- `{flags}` (`string`) See tag-function

Return:
(`table[]|vim.NIL`) tags A list of matching tags, or `vim.NIL` to use the built-in tags.

Lua module: vim.lsp.buf lsp-buf

The `vim.lsp.buf_…` functions perform operations for LSP clients attached to the current buffer.

vim.lsp.ListOpts

Fields:

- `{loclist}?` (`boolean`) Whether to use the location-list or the quickfix list in the default handler.

```
vim.lsp.buf.definition({ loclist = true })
vim.lsp.buf.references(nil, { loclist = false })
```

- `{on_list}?` (`fun(what: vim.fn.setqflist.what)`) list-handler replacing the default handler. Called for any non-empty result. The default handler populates the quickfix (or location) list with the result. If there is a single result (and the method is not `implementation` or `references`), it pushes a tag onto the tagstack and jumps to the result. For example, when `loclist == false` (the default), the handler is equivalent to:

```
local function on_list(what)
  vim.fn.setqflist({}, ' ', what)
  if
    #what.items == 1
    and what.context.method ~= 'textDocument/implementation'
    and what.context.method ~= 'textDocument/references'
  then
    local tagstack = { { tagname = tagname, from = from } }
    vim.fn.settagstack(vim.fn.win_getid(win), { items = tagstack }, 't')
    vim.cmd('cfirst')
  else
    vim.cmd('botright copen')
  end
end

vim.lsp.buf.definition({ on_list = on_list })
vim.lsp.buf.references(nil, { on_list = on_list })
```

The list can be transformed before it is shown. For example, to remove duplicate locations returned by multiple clients:

```
local function on_list(what)
  vim.list.unique(what.items, function(item)
    return ('%s\0%d\0%d\0%d\0%d'):format(
      item.filename or '',
      item.lnum or 0,
      item.col or 0,
      item.end_lnum or 0,
      item.end_col or 0
    )
  end)
  vim.fn.setqflist({}, ' ', what)
  vim.cmd('botright copen')
end

vim.lsp.buf.definition({ on_list = on_list })
vim.lsp.buf.references(nil, { on_list = on_list })
```

See setqflist-what for the structure of the `what` parameter.

- `{pos}`? (`vim.Pos`, default: cursor position) Position on a buffer to request.

vim.lsp.buf.hover.Opts Extends: vim.lsp.util.open_floating_preview.Opts

Fields:

- `{silent}`? (`boolean`)

vim.lsp.buf.signature_help.Opts Extends: vim.lsp.util.open_floating_preview.Opts

Fields:

- `{silent}`? (`boolean`)

vim.lsp.buf.add_workspace_folder()

add_workspace_folder(`{workspace_folder}`) Add the folder at path to the workspace folders. If `{path}` is not provided, the user will be prompted for a path using input().

Parameters:

- `{workspace_folder}` (`string?`)

clear_references() vim.lsp.buf.clear_references()

Removes document highlights from current buffer.

code_action(`{opts}`) vim.lsp.buf.code_action()

Selects a code action (LSP: "textDocument/codeAction" request) available at cursor position.

Parameters:

- `{opts}` (`table?`) A table with the following fields:
  - `{apply}`? (`boolean`) When set to `true`, and there is just one remaining action (after filtering), the action is applied without user query.
  - `{context}`? (`table`) Corresponds to `lsp.CodeActionContext`, but all fields are optional:
    - `{diagnostics}`? (`lsp.Diagnostic[]`) Inferred from the current position if not provided.
    - `{only}`? (`lsp.CodeActionKind[]`) `CodeActionKinds` used to filter the code actions. Most servers support values like "refactor" or "quickfix".
    - `{triggerKind}`? (`lsp.CodeActionTriggerKind`) Why code actions were requested.
  - `{filter}`? (`fun(x: lsp.CodeAction|lsp.Command, client_id: integer):boolean`) Predicate taking a code action or command and the provider's ID. If it returns false, the action is filtered out.
  - `{range}`? (`{start: integer[], end: integer[]}`) Range for which code actions should be requested. If in visual mode this defaults to the active selection. Table must contain `start` and `end` keys with `{row,col}` tuples using mark-like indexing. See api-indexing

See also:

- https://microsoft.github.io/language-server-protocol/specifications/specification-current/#textDocument_codeAction
- vim.lsp.protocol.CodeActionTriggerKind

declaration(`{opts}`) vim.lsp.buf.declaration()

Jumps to the declaration of the symbol under the cursor.

Note:

- Many servers do not implement this method. Generally, see vim.lsp.buf.definition() instead.

Parameters:

- `{opts}` (`vim.lsp.ListOpts?`)

definition({opts}) vim.lsp.buf.definition()
Jumps to the definition of the symbol under the cursor.

Parameters:
- {opts} (vim.lsp.ListOpts?)

document_highlight() vim.lsp.buf.document_highlight()
Send request to the server to resolve document highlights for the current text document position. This request can be triggered by a key mapping or by events such as CursorHold, e.g.:

```
autocmd CursorHold  <buffer> lua vim.lsp.buf.document_highlight()
autocmd CursorHoldI <buffer> lua vim.lsp.buf.document_highlight()
autocmd CursorMoved <buffer> lua vim.lsp.buf.clear_references()
```

Note: Usage of vim.lsp.buf.document_highlight() requires the following highlight groups to be defined or you won't be able to see the actual highlights. hl-LspReferenceText hl-LspReferenceRead hl-LspReferenceWrite

document_symbol({opts}) vim.lsp.buf.document_symbol()
Lists all symbols in the current buffer in the location-list.

Parameters:
- {opts} (vim.lsp.ListOpts?)

format({opts}) vim.lsp.buf.format()
Formats a buffer using the attached (and optionally filtered) language server clients.

Parameters:
- {opts} (table?) A table with the following fields:
  - {async}? (boolean, default: false) If true the method won't block. Editing the buffer while formatting asynchronous can lead to unexpected changes.
  - {bufnr}? (integer, default: current buffer) Restrict formatting to the clients attached to the given buffer.
  - {filter}? (fun(client: vim.lsp.Client): boolean?) Predicate used to filter clients. Receives a client as argument and must return a boolean. Clients matching the predicate are included. Example:

    ```
    -- Never request typescript-language-server for formatting
    vim.lsp.buf.format {
      filter = function(client) return client.name ~= "ts_ls" end
    }
    ```

  - {formatting_options}? (lsp.FormattingOptions) Can be used to specify FormattingOptions. Some unspecified options will be automatically derived from the current Nvim options. See https://microsoft.github.io/language-server-protocol/specification/#formattingOptions
  - {id}? (integer) Restrict formatting to the client with ID (client.id) matching this field.
  - {name}? (string) Restrict formatting to the client with name (client.name) matching this field.
  - {range}? ({start:[integer,integer],end:[integer, integer]}|{start:[integer,integer],end:[integer,integer]}[], default: current selection in visual mode, nil in other modes, formatting the full buffer) Range to format. Table must contain start and end keys with {row,col} tuples using (1,0) indexing. The end column can be -1 to format through the end of the line. Can also be a list of tables that contain start and end keys as described above, in which case textDocument/rangesFormatting support is required.
  - {timeout_ms}? (integer, default: 1000 for synchronous requests) Time in milliseconds allowed for each formatting request. On timeout, cancels the request and continues with the next client. With async=true, there is no timeout unless this option is set. Responses received after the request times out are ignored.

hover({config}) vim.lsp.buf.hover()
Displays hover information about the symbol under the cursor in a floating window. The window will be dismissed on cursor move. Calling the function twice will jump into the floating window (thus by default, "KK" will open the hover window and focus it). In the floating window, all commands and mappings are available as usual, except that "q" dismisses the window. You can scroll the contents the same as you would any other buffer.

Note: to disable hover highlights, add the following to your config:

```
vim.api.nvim_create_autocmd('ColorScheme', {
  callback = function()
    vim.api.nvim_set_hl(0, 'LspReferenceTarget', {})
  end,
})
```

Parameters:
- {config} (vim.lsp.buf.hover.Opts?)

implementation({opts}) vim.lsp.buf.implementation()
Lists all the implementations for the symbol under the cursor in the quickfix window.

Parameters:
- {opts} (vim.lsp.ListOpts?)

incoming_calls() vim.lsp.buf.incoming_calls()
Lists all the call sites of the symbol under the cursor in the quickfix window. If the symbol can resolve to multiple items, the user can pick one in the inputlist().

list_workspace_folders() vim.lsp.buf.list_workspace_folders()
List workspace folders.

outgoing_calls() vim.lsp.buf.outgoing_calls()
Lists all the items that are called by the symbol under the cursor in the quickfix window. If the symbol can resolve to multiple items, the user can pick one in the inputlist().

references({context}, {opts}) vim.lsp.buf.references()
Lists all the references to the symbol under the cursor in the quickfix window.

Parameters:
- {context} (lsp.ReferenceContext?) Context for the request
- {opts} (vim.lsp.ListOpts?)

See also:
- https://microsoft.github.io/language-server-protocol/specifications/specification-current/#textDocument_references

vim.lsp.buf.remove_workspace_folder()
remove_workspace_folder({workspace_folder}) Remove the folder at path from the workspace folders. If {path} is not provided, the user will be prompted for a path using input().

Parameters:
- {workspace_folder} (string?)

rename({new_name}, {opts}) vim.lsp.buf.rename()
Renames all references to the symbol under the cursor.

Parameters:
- {new_name} (string?) If not provided, the user will be prompted for a new name using vim.ui.input().
- {opts} (table?) Additional options:
  - {bufnr}? (integer) (default: current buffer)
  - {filter}? (fun(client: vim.lsp.Client): boolean?) Predicate used to filter clients. Receives a client as argument and must return a boolean. Clients matching the predicate are included.
  - {name}? (string) Restrict clients used for rename to ones where client.name matches this field.

vim.lsp.buf.selection_range()
selection_range({direction}, {timeout_ms}) lsp-incremental-selection

Expands or contracts a Visual selection at cursor, based on ranges given by LSP. The direction parameter specifies the number of times to expand the selection. Negative values will shrink the selection.

Parameters:
- {direction} (integer)
- {timeout_ms} (integer?) (default: 1000) Maximum time (milliseconds) to wait for a result.

signature_help({config}) vim.lsp.buf.signature_help()
Displays signature information about the symbol under the cursor in a floating window. Allows cycling through signature overloads with <C-s>, which can be remapped via <Plug>(nvim.lsp.ctrl-s)

Example:

```
vim.keymap.set('n', '<C-b>', '<Plug>(nvim.lsp.ctrl-s)')
```

Parameters:
- {config} (vim.lsp.buf.signature_help.Opts?)

type_definition({opts}) vim.lsp.buf.type_definition()
Jumps to the definition of the type of the symbol under the cursor.

Parameters:
- {opts} (vim.lsp.ListOpts?)

typehierarchy({kind}) vim.lsp.buf.typehierarchy()
Lists all the subtypes or supertypes of the symbol under the cursor in the quickfix window. If the symbol can resolve to multiple items, the user can pick one using vim.ui.select().

Parameters:
- {kind} ("subtypes"|"supertypes")

workspace_diagnostics({opts}) vim.lsp.buf.workspace_diagnostics()
Request workspace-wide diagnostics.

Parameters:
- {opts} (table?) A table with the following fields:
  - {client_id}? (integer) Only request diagnostics from the indicated client. If nil, the request is sent to all clients.

See also:
- https://microsoft.github.io/language-server-protocol/specifications/specification-current/#workspace_diagnostic

workspace_symbol({query}, {opts}) vim.lsp.buf.workspace_symbol()
Lists all symbols in the current workspace in the quickfix window.

The list is filtered against {query}; if the argument is omitted from the call, the user is prompted to enter a string on the command line. An empty string means no filtering is done.

Parameters:
- {query} (string?) optional
- {opts} (vim.lsp.ListOpts?)

Lua module: vim.lsp.client lsp-client

vim.lsp.Client

Fields:
- {attached_buffers} (table<integer,string>) Each buffer's last used languageId.
- {cancel_request} (fun(self: vim.lsp.Client, id: integer): boolean) See Client:cancel_request().

- {capabilities} (lsp.ClientCapabilities) Capabilities provided by the client (editor or tool), at startup.
- {commands} (table<string,fun(command: lsp.Command, ctx: table)>) Client commands. See vim.lsp.ClientConfig.
- {config} (vim.lsp.ClientConfig) Copy of the config passed to vim.lsp.start().
- {dynamic_capabilities} (lsp.DynamicCapabilities) Capabilities provided at runtime (after startup).
- {exec_cmd} (fun(self: vim.lsp.Client, cmd: lsp.Command, context: {bufnr?: integer}?, handler: lsp.Handler?)) See Client:exec_cmd().
- {exit_timeout} (integer|boolean, default: false) See vim.lsp.ClientConfig.
- {flags} (table) Experimental client flags:
  - {allow_incremental_sync}? (boolean, default: true) Allow using incremental sync for buffer edits
  - {debounce_text_changes}? (integer, default: 150) Debounce didChange notifications to the server by the given number in milliseconds.
- {get_language_id} (fun(bufnr: integer, filetype: string): string) See vim.lsp.ClientConfig.
- {handlers} (table<string,lsp.Handler>) See vim.lsp.ClientConfig.
- {id} (integer) The id allocated to the client.
- {initialized} (true?)
- {is_stopped} (fun(self: vim.lsp.Client): boolean) See Client:is_stopped().
- {name} (string) See vim.lsp.ClientConfig.
- {notify} (fun(self: vim.lsp.Client, method: string, params: table?, bufnr: integer?): boolean) See Client:notify().
- {offset_encoding} ('utf-8'|'utf-16'|'utf-32') See vim.lsp.ClientConfig.
- {on_attach} (fun(self: vim.lsp.Client, bufnr: integer)) See Client:on_attach().
- {progress} (vim.lsp.Client.Progress) A ring buffer (vim.ringbuf()) containing progress messages sent by the server.
- {request} (fun(self: vim.lsp.Client, method: string, params: table?, handler: lsp.Handler?, bufnr: integer?): boolean, integer?) See Client:request().
- {request_sync} (fun(self: vim.lsp.Client, method: string, params: table, timeout_ms: integer?, bufnr: integer?): {err: lsp.ResponseError?, result:any}?, string?) See Client:request_sync().
- {requests} (table<integer,{ type: string, bufnr: integer, method: vim.lsp.protocol.Method}?>) The current pending requests in flight to the server. Entries are key-value pairs with the key being the request id while the value is a table with type, bufnr, and method key-value pairs. type is either "pending" for an active request, or "cancel" for a cancel request. It will be "complete" ephemerally while executing LspRequest autocmds when replies are received from the server.
- {root_dir} (string?) See vim.lsp.ClientConfig.
- {rpc} (vim.lsp.rpc.Client) RPC client object, for low level interaction with the client. See vim.lsp.rpc.start().
- {server_capabilities} (lsp.ServerCapabilities) Response from the server sent on initialize describing the server's capabilities.
- {server_info} (lsp.ServerInfo?) Response from the server sent on initialize describing server information (e.g. version).
- {settings} (lsp.LSPObject) See vim.lsp.ClientConfig.
- {stop} (fun(self: vim.lsp.Client, force: integer|boolean?)) See Client:stop().
- {supports_method} (fun(self: vim.lsp.Client, method: string|string, bufnr: integer?): boolean) See Client:supports_method().
- {workspace_folders} (lsp.WorkspaceFolder[]?) See vim.lsp.ClientConfig.

vim.lsp.Client.Progress Extends: vim.Ringbuf

Fields:
- {pending} (table<lsp.ProgressToken,lsp.LSPAny>)

vim.lsp.ClientConfig

Fields:
- {before_init}? (fun(params: lsp.InitializeParams, config: vim.lsp.ClientConfig)) Callback which can modify parameters before they are sent to the server. Invoked before LSP "initialize" phase (after cmd is invoked), where params is the parameters being sent to the server and config is the config passed to vim.lsp.start().

Hint: use vim.tbl_deep_extend() to set nested fields easily.

```
before_init = function(_, config)
  config.settings = vim.tbl_deep_extend('force',
    config.settings,
    { tailwindCSS = { experimental = { configFile = find_tailwind_global_css() } } }
  )
end
```

- {capabilities}? (lsp.ClientCapabilities) Map overriding the default capabilities defined by vim.lsp.protocol.make_client_capabilities(), passed to the language server on initialization. Hint: use make_client_capabilities() and modify its result.
  - **Note:** To send an empty dictionary use vim.empty_dict(), else it will be encoded as an array.
- {cmd} (string[]|fun(dispatchers: vim.lsp.rpc.Dispatchers, config: vim.lsp.ClientConfig): vim.lsp.rpc.Client) Command string[] that launches the language server (treated as in jobstart(), must be absolute or on $PATH, shell constructs like "~" are not expanded), or function that creates an RPC client (or an in-process lsp-server). Function receives a dispatchers table and the resolved config, and must return an object in the form of vim.lsp.rpc.Client.

Example (list):

```
cmd = { 'flow', 'lsp' }
```

Example (function):

```
cmd = function(dispatchers, config)
  local cmd = vim.fn.executable('flow') == 1 and { 'flow', 'lsp' } or { 'npx', '--no-install', 'flow', 'lsp' }
  return vim.lsp.rpc.start(cmd, dispatchers)
end
```

- See vim.lsp.rpc.Client.
- For TCP there is a builtin RPC client factory: vim.lsp.rpc.connect()

- {cmd_cwd}? (string) Directory where the cmd process is launched when cmd is a string array. Defaults to root_dir when available, then the current working directory.
- {cmd_env}? (table) Environment variables passed to the LSP process on spawn. Non-string values are coerced to string. Example:

```
{ PORT = 8080, HOST = '0.0.0.0' }
```

- {commands}? (table<string,fun(command: lsp.Command, ctx: table)>) Map of client-defined commands overriding the global vim.lsp.commands.
- {detached}? (boolean, default: true) Daemonize the server process so that it runs in a separate process group from Nvim. Nvim will shutdown the process on exit, but if Nvim fails to exit cleanly this could leave behind orphaned server processes.
- {exit_timeout}? (integer|boolean, default: false) Decides if/when to force-stop the server after sending the "shutdown" request. See Client:stop(). **Note:** when Nvim itself is exiting,
  - false: Nvim will not force-stop LSP server(s).
  - true: Nvim will force-stop LSP server(s) that did not comply with the "shutdown" request.
  - number: Nvim will wait up to exit_timeout milliseconds before performing force-stop.
- {flags}? (table) Experimental client flags:
  - {allow_incremental_sync}? (boolean, default: true) Allow using incremental sync for buffer edits
  - {debounce_text_changes}? (integer, default: 150) Debounce didChange notifications to the server by the given number in milliseconds.
- {get_language_id}? (fun(bufnr: integer, filetype: string): string) Language ID as string. Defaults to the buffer filetype.
- {handlers}? (table<string,function>) Map of LSP method names to lsp-handlers.
- {init_options}? (lsp.LSPObject) Values to pass in the initialization request as initializationOptions. See initialize in the LSP spec.
- {name}? (string, default: client-id) Name in logs and user messages.
- {offset_encoding}? ('utf-8'|'utf-16'|'utf-32') Called "position encoding" in LSP spec. The encoding that the LSP server expects, used for communication. Not validated. Can be modified in on_init before text is sent to the server.
- {on_attach}? (elem_or_list<fun(client: vim.lsp.Client, bufnr: integer)>) Callback invoked when client attaches to a buffer.
- {on_error}? (fun(code: integer, err: string)) Callback invoked when the client operation throws an error. code is a number describing the error. Other arguments may be passed depending on the error kind. See vim.lsp.rpc.client_errors for possible errors. Use vim.lsp.rpc.client_errors[code] to get human-friendly name.
- {on_exit}? (elem_or_list<fun(code: integer, signal: integer, client_id: integer)>) Callback invoked on client exit.
  - code: exit code of the process
  - signal: number describing the signal used to terminate (if any)
  - client_id: client handle
- {on_init}? (elem_or_list<fun(client: vim.lsp.Client, init_result: lsp.InitializeResult)>) Callback invoked after LSP "initialize", where result is a table of capabilities and anything else the server may send. For example, clangd sends init_result.offsetEncoding if capabilities.offsetEncoding was sent to it. You can only modify the client.offset_encoding here before any notifications are sent.
- {root_dir}? (string) Directory where the LSP server will base its workspaceFolders, rootUri, and rootPath on initialization.
- {settings}? (lsp.LSPObject) Map of language server-specific settings, decided by the client. Sent to the LS if requested via workspace/configuration. Keys are case-sensitive.
- {trace}? ('off'|'messages'|'verbose', default: "off") Passed directly to the language server in the initialize request. Invalid/empty values will
- {workspace_folders}? (lsp.WorkspaceFolder[]) List of workspace folders passed to the language server. For backwards compatibility rootUri and rootPath are derived from the first workspace folder in this list. Can be null if the client supports workspace folders but none are configured. See workspaceFolders in LSP spec.
- {workspace_required}? (boolean, default: false) Server requires a workspace (no "single file" support). **Note:** Without a workspace, cross-file features (navigation, hover) may or may not work depending on the language server, even if the server doesn't require a workspace.

Client:cancel_request({id}) Client:cancel_request()
Cancels a request with a given request id.

Parameters:
- {id} (integer) id of request to cancel

Return:
(boolean) status indicating if the notification was successful.

See also:
- Client:notify()

Client:exec_cmd({cmd}, {context}, {handler}) Client:exec_cmd()
Execute a lsp command, either via client command function (if available) or via workspace/executeCommand (if supported by the server)

Parameters:
- {cmd} (lsp.Command)
- {context} ({bufnr?: integer}?)
- {handler} (lsp.Handler?) only called if a server command

Client:is_stopped() Client:is_stopped()
Checks whether a client is stopped.

Return:
(boolean) true if client is stopped or in the process of being stopped; false otherwise

Client:notify({method}, {params}, {bufnr}) Client:notify()
Sends a notification to an LSP server.

Parameters:
- {method} (string) LSP method name.
- {params} (table?) LSP request params.
- {bufnr} (integer?) Buffer associated with notification.

Return:
(boolean) status indicating if the notification was successful. If it is false, then the client has shutdown.

Client:on_attach({bufnr}) Client:on_attach()
Runs the on_attach function from the client's config if it was defined. Useful for buffer-local setup.

Parameters:
- {bufnr} (integer) Buffer number

Client:request()

Client:request({method}, {params}, {handler}, {bufnr}) Sends a request to the server.

This is a thin wrapper around {client.rpc.request} with some additional checks for capabilities and handler availability.

Parameters:
- {method} (string) LSP method name.
- {params} (table?) LSP request params.
- {handler} (lsp.Handler?) Response lsp-handler for this method.
- {bufnr} (integer?) (default: 0) Buffer handle, or 0 for current.

Return (multiple):
(boolean) status indicates whether the request was successful. If it is false, then it will always be false (the client has shutdown).
(integer?) request_id Can be used with Client:cancel_request(). nil is request failed.

See also:
- vim.lsp.buf_request_all()

Client:request_sync()

Client:request_sync({method}, {params}, {timeout_ms}, {bufnr}) Sends a request to the server and synchronously waits for the response.

This is a wrapper around Client:request()

Parameters:
- {method} (string) LSP method name.
- {params} (table) LSP request params.
- {timeout_ms} (integer?) Maximum time in milliseconds to wait for a result. Defaults to 1000
- {bufnr} (integer?) (default: 0) Buffer handle, or 0 for current.

Return (multiple):
({err: lsp.ResponseError?, result:any}?) result and err from the lsp-handler. nil is the request was unsuccessful (string?) err On timeout, cancel or error, where err is a string describing the failure reason.

See also:
- vim.lsp.buf_request_sync()

Client:stop({force}) Client:stop()
Stops a client, optionally with force after a timeout.

By default this sends a "shutdown" request to the server, escalating to force-stop if the server has not exited after self.exit_timeout milliseconds (unless exit_timeout=false). Calling stop() on a client that was previously requested to shutdown, will escalate to force-stop immediately, regardless of force (or self.exit_timeout if force=nil).

Note: Forcing shutdown while a server is busy writing out project or index files can lead to file corruption.

Parameters:
- {force} (integer|boolean?) (default: self.exit_timeout) Decides whether to force-stop the server.
  - false: Do not force-stop after "shutdown" request.
  - nil: Defaults to exit_timeout from vim.lsp.ClientConfig.
  - true: Force-stop after "shutdown" request.
  - number: Wait up to force milliseconds before force-stop.

Client:supports_method({method}, {bufnr}) Client:supports_method()
Checks if a client supports a given method. Always returns true for unknown off-spec methods.

Note: Some language server capabilities can be file specific.

Parameters:
- {method} (string|string)
- {bufnr} (integer?)

Return:
(boolean)

Lua module: vim.lsp.codelens lsp-codelens

enable({enable}, {filter}) vim.lsp.codelens.enable()
Enables or disables code lens for the {filter}ed scope.

To "toggle", pass the inverse of is_enabled():

```
vim.lsp.codelens.enable(not vim.lsp.codelens.is_enabled())
```

To run a code lens, see vim.lsp.codelens.run().

Parameters:
- {enable} (boolean?) true/nil to enable, false to disable
- {filter} (table?) Optional filters kwargs,
  - {bufnr}? (integer, default: all) Buffer number, or 0 for current buffer, or nil for all.
  - {client_id}? (integer, default: all) Client ID, or nil for all.

get({filter}) vim.lsp.codelens.get()
Get all code lenses in the {filter}ed scope.

Parameters:
- {filter} (table?) Optional filters kwargs:
  - {bufnr}? (integer, default: 0) Buffer handle, or 0 for current.
  - {client_id}? (integer, default: all) Client ID, or nil for all.

Overloads:
- fun(filter: integer): lsp.CodeLens[]

Return:
(table[]) A list of objects with the following fields:
  - {client_id} (integer)
  - {lens} (lsp.CodeLens)

is_enabled({filter}) vim.lsp.codelens.is_enabled()
Query whether code lens is enabled in the {filter}ed scope

Parameters:
- {filter} (table?) Optional filters kwargs,
  - {bufnr}? (integer, default: all) Buffer number, or 0 for current buffer, or nil for all.
  - {client_id}? (integer, default: all) Client ID, or nil for all.

Return:
(boolean) whether code lens is enabled.

run({opts}) vim.lsp.codelens.run()
Run code lens at the current cursor position.

Parameters:
- {opts} (table?) Optional parameters kwargs:
  - {client_id}? (integer, default: all) Client ID, or nil for all.

Lua module: vim.lsp.completion lsp-completion

The vim.lsp.completion module enables insert-mode completion driven by an LSP server. Call enable() to make it available through Nvim builtin completion (via the CompleteDone event). Specify autotrigger=true to activate "auto-completion" when you type any of the server-defined triggerCharacters. Use CTRL-Y to select an item from the completion menu. complete_CTRL-Y

Example: activate LSP-driven auto-completion:

```
-- Works best if 'completeopt' has "noselect".
-- Use CTRL-Y to select an item. |complete_CTRL-Y|
vim.cmd[[set completeopt+=menuone,noselect,popup]]
vim.lsp.start({
  name = 'ts_ls',
  cmd = …,
  on_attach = function(client, bufnr)
    vim.lsp.completion.enable(true, client.id, bufnr, {
      autotrigger = true,
      convert = function(item)
        return { abbr = item.label:gsub('%b()', '') }
      end,
    })
  end,
})
```

lsp-autocompletion

The LSP triggerCharacters field decides when to trigger autocompletion. If you want to trigger on EVERY keypress you can either:
- Extend client.server_capabilities.completionProvider.triggerCharacters on LspAttach, before you call vim.lsp.completion.enable(… {autotrigger=true}). See the lsp-attach example.
- Call vim.lsp.completion.get() from an InsertCharPre autocommand.

If the server provides commitCharacters for a completion item, typing one of those characters while the item is selected accepts the completion and then inserts the character. To disable this:

```
vim.lsp.completion.enable(true, client_id, bufnr, { commit_characters = false })
```

vim.lsp.completion.enable()
enable({enable}, {client_id}, {bufnr}, {opts}) Enables or disables completions from the given language client in the given buffer. Effects of enabling completions are:
- Calling vim.lsp.completion.get() uses the enabled clients to retrieve completion candidates.
- Selecting a completion item shows a preview popup ("completionItem/resolve") if 'completeopt' has "popup".
- Accepting a completion item using <c-y> applies side effects like expanding snippets, text edits (e.g. insert import statements) and executing associated commands. This works for completions triggered via autotrigger, 'omnifunc' or vim.lsp.completion.get().

Examples: lsp-attach lsp-completion

Note:
- vim.lsp.omnifunc() (i_CTRL-X_CTRL-O) queries every client that advertises completion, including clients that were disabled by enable(false). To suppress completions for a client, clear its capability on LspAttach: client.server_capabilities.completionProvider = nil.
- Behavior of autotrigger=true is controlled by the LSP triggerCharacters field. You can override it on LspAttach, see lsp-autocompletion.

Parameters:

- {enable} (boolean) True to enable, false to disable
- {client_id} (integer) Client ID
- {bufnr} (integer) Buffer handle, or 0 for the current buffer
- {opts} (table?) A table with the following fields:
  - {autotrigger}? (boolean) (default: false) When true, completion triggers automatically based on the server's `triggerCharacters`.
  - {cmp}? (fun(a: table, b: table): boolean) Comparator for sorting merged completion items from all servers.
  - {commit_characters}? (boolean) (default: true) When false, commit characters are ignored.
  - {convert}? (fun(item: lsp.CompletionItem): table) Transforms an LSP CompletionItem to complete-items.

get({opts}) vim.lsp.completion.get()
Triggers LSP completion once in the current buffer, if LSP completion is enabled (see lsp-attach lsp-completion).

Use CTRL-Y to select an item from the completion menu. complete_CTRL-Y

To invoke manually with CTRL-space, use this mapping:

```
-- Use CTRL-space to trigger LSP completion.
-- Use CTRL-Y to select an item. |complete_CTRL-Y|
vim.keymap.set('i', '<c-space>', function()
  vim.lsp.completion.get()
end)
```

Parameters:
- {opts} (table?) A table with the following fields:
  - {client_id}? (integer|integer[]) Only request from these clients. Defaults to all clients with completion enabled.
  - {ctx}? (lsp.CompletionContext) Completion context. Defaults to a trigger kind of `invoked`.

Lua module: vim.lsp.diagnostic lsp-diagnostic

This module provides functionality for requesting LSP diagnostics for a document/workspace and populating them using vim.Diagnostics. `DiagnosticRelatedInformation` is supported: it is included in the window shown by vim.diagnostic.open_float(). When the cursor is on a line with related information, gf jumps to the problem location.

from({diagnostics}) vim.lsp.diagnostic.from()
Converts the input `vim.Diagnostics` to LSP diagnostics.

Parameters:
- {diagnostics} (vim.Diagnostic[])

Return:
(lsp.Diagnostic[])

vim.lsp.diagnostic.get_namespace()
get_namespace({client_id}, {is_pull}, {pull_id}) Get the diagnostic namespace associated with an LSP client vim.diagnostic for diagnostics

Parameters:
- {client_id} (integer) The id of the LSP client
- {is_pull} (boolean?) Whether the namespace is for a pull or push client. Defaults to `false` (push).
- {pull_id} (string?) (default: nil) Optional identifier for pull diagnostic providers. Only used if `is_pull` is `true`.

vim.lsp.diagnostic.on_diagnostic()
on_diagnostic({error}, {result}, {ctx}) lsp-handler for the method "textDocument/diagnostic"

See vim.diagnostic.config() for configuration options.

Parameters:
- {error} (lsp.ResponseError?)
- {result} (lsp.DocumentDiagnosticReport)
- {ctx} (lsp.HandlerContext)

vim.lsp.diagnostic.on_publish_diagnostics()
on_publish_diagnostics({_}, {params}, {ctx}) lsp-handler for the method "textDocument/publishDiagnostics"

See vim.diagnostic.config() for configuration options.

Parameters:
- {params} (lsp.PublishDiagnosticsParams)
- {ctx} (lsp.HandlerContext)

Lua module: vim.lsp.document_color lsp-document_color

This module provides LSP support for highlighting color references in a document. Highlighting is enabled by default.

color_presentation() vim.lsp.document_color.color_presentation()
Select from a list of presentations for the color under the cursor.

enable({enable}, {filter}, {opts}) vim.lsp.document_color.enable()
Enables or disables document color highlighting for the {filter}ed scope.

To "toggle", pass the inverse of `is_enabled()`:

```
vim.lsp.document_color.enable(not vim.lsp.document_color.is_enabled())
```

Parameters:
- {enable} (boolean?) True to enable, false to disable. (default: `true`)
- {filter} (table?) Optional filters kwargs,

- {bufnr}? (integer, default: all) Buffer number, or 0 for current buffer, or nil for all.
- {client_id}? (integer, default: all) Client ID, or nil for all.

- {opts} (table?) A table with the following fields:
  - {style}? ('background'|'foreground'|'virtual'|string|fun(bufnr: integer, range: vim.Range, hex_code: string)) Highlight style. It can be one of the pre-defined styles, a string to be used as virtual text, or a function that receives the buffer handle, the range (start line, start col, end line, end col) and the resolved hex color. (default: 'background')

is_enabled({filter}) vim.lsp.document_color.is_enabled()
Query whether document colors are enabled in the {filter}ed scope.

Parameters:
- {filter} (table?) Optional filters kwargs,
  - {bufnr}? (integer, default: all) Buffer number, or 0 for current buffer, or nil for all.
  - {client_id}? (integer, default: all) Client ID, or nil for all.

Return:
(boolean)

Lua module: vim.lsp.inlay_hint lsp-inlay_hint

enable({enable}, {filter}) vim.lsp.inlay_hint.enable()
Enables or disables inlay hints for the {filter}ed scope.

To "toggle", pass the inverse of is_enabled():

```
vim.lsp.inlay_hint.enable(not vim.lsp.inlay_hint.is_enabled())
```

Attributes:
Since: 0.10.0

Parameters:
- {enable} (boolean?) true/nil to enable, false to disable
- {filter} (table?) Optional filters kwargs,
  - {bufnr}? (integer, default: all) Buffer number, or 0 for current buffer, or nil for all.
  - {client_id}? (integer, default: all) Client ID, or nil for all.

get({filter}) vim.lsp.inlay_hint.get()
Get the list of inlay hints, (optionally) restricted by buffer or range.

Example usage:

```
local hint = vim.lsp.inlay_hint.get({ bufnr = 0 })[1] -- 0 for current buffer

local client = vim.lsp.get_client_by_id(hint.client_id)
local resp = client:request_sync('inlayHint/resolve', hint.inlay_hint, 100, 0)
local resolved_hint = assert(
  resp and resp.result,
  resp and resp.err and vim.lsp.rpc.format_rpc_error(resp.err) or 'request failed'
)
vim.lsp.util.apply_text_edits(resolved_hint.textEdits, 0, client.encoding)

location = resolved_hint.label[1].location
client:request('textDocument/hover', {
  textDocument = { uri = location.uri },
  position = location.range.start,
})
```

Attributes:
Since: 0.10.0

Parameters:
- {filter} (table?) Optional filters kwargs:
  - {bufnr} (integer?)
  - {range} (lsp.Range?)

Return:
(table[]) A list of objects with the following fields:
- {bufnr} (integer)
- {client_id} (integer)
- {inlay_hint} (lsp.InlayHint)

is_enabled({filter}) vim.lsp.inlay_hint.is_enabled()
Query whether inlay hint is enabled in the {filter}ed scope

Attributes:
Since: 0.10.0

Parameters:
- {filter} (table?) Optional filters kwargs,
  - {bufnr}? (integer, default: all) Buffer number, or 0 for current buffer, or nil for all.
  - {client_id}? (integer, default: all) Client ID, or nil for all.

Return:
(boolean)

Lua module: vim.lsp.inline_completion

lsp-inline_completion

This module provides the LSP "inline completion" feature, for completing multiline text (e.g., whole methods) instead of just a word or line, which may result in "syntactically or semantically incorrect" code. Unlike regular completion, this is typically presented as overlay text instead of a menu of completion candidates.

LSP spec: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.18/specification/#textDocument_inlineCompletion

To try it out, here is a quickstart example using Copilot: lsp-copilot

1. Install Copilot:

```
npm install --global @github/copilot-language-server
```

1. Define a config, (or copy `lsp/copilot.lua` from https://github.com/neovim/nvim-lspconfig):

```
 vim.lsp.config('copilot', {
  cmd = { 'copilot-language-server', '--stdio' },
  root_markers = { '.git' },
   init_options = {
     editorInfo = {
       name = 'Neovim', version = tostring(vim.version()) },
       editorPluginInfo = { name = 'Neovim', version = tostring(vim.version()) },
     },
})
```

1. Activate the config:

```
vim.lsp.enable('copilot')
```

1. Sign in to Copilot, or use the `:LspCopilotSignIn` command from https://github.com/neovim/nvim-lspconfig
2. Enable inline completion:

```
vim.lsp.inline_completion.enable()
```

1. Set a keymap for `vim.lsp.inline_completion.get()` and invoke the keymap.

vim.lsp.inline_completion.Item

Fields:

- `{client_id}` (`integer`) Client ID
- `{command}?` (`lsp.Command`) Corresponding server command.
- `{insert_text}` (`string|lsp.StringValue`) The text to be inserted, can be a snippet.
- `{range}?` (`vim.Range`) Which range it be applied.

enable({enable}, {filter}) vim.lsp.inline_completion.enable()

Enables or disables inline completion for the `{filter}`ed scope, inline completion will automatically be refreshed when you are in insert mode.

To "toggle", pass the inverse of `is_enabled()`:

```
vim.lsp.inline_completion.enable(not vim.lsp.inline_completion.is_enabled())
```

Parameters:

- `{enable}` (`boolean?`) true/nil to enable, false to disable
- `{filter}` (`table?`) Optional filters kwargs,
  - `{bufnr}?` (`integer`, default: all) Buffer number, or 0 for current buffer, or nil for all.
  - `{client_id}?` (`integer`, default: all) Client ID, or nil for all.

get({opts}) vim.lsp.inline_completion.get()

Accept the currently displayed completion candidate to the buffer.

It returns false when no candidate can be accepted, so you can use the return value to implement a fallback:

```
 vim.keymap.set('i', '<Tab>', function()
  if not vim.lsp.inline_completion.get() then
    return '<Tab>'
  end
end, { expr = true, desc = 'Accept the current inline completion' })
```

Parameters:

- `{opts}` (`table?`) A table with the following fields:
  - `{bufnr}?` (`integer`, default: 0) Buffer handle, or 0 for current.
  - `{on_accept}?` (`fun(item: vim.lsp.inline_completion.Item): vim.lsp.inline_completion.Item?`) A callback triggered when a completion item is accepted. You can use it to modify the completion item that is about to be accepted and return it to apply the changes, or return `nil` to prevent the changes from being applied to the buffer so you can implement custom behavior.

Return:

(`boolean`) `true` if a completion was applied, else `false`.

is_enabled({filter}) vim.lsp.inline_completion.is_enabled()

Query whether inline completion is enabled in the `{filter}`ed scope

Parameters:

- `{filter}` (`table?`) Optional filters kwargs,
  - `{bufnr}?` (`integer`, default: all) Buffer number, or 0 for current buffer, or nil for all.
  - `{client_id}?` (`integer`, default: all) Client ID, or nil for all.

select({opts}) vim.lsp.inline_completion.select()
Switch between available inline completion candidates.

Parameters:
- {opts} (table?) A table with the following fields:
  - {bufnr}? (integer) (default: current buffer)
  - {count}? (integer, default: v:count1) The number of candidates to move by. A positive integer moves forward by {count} candidates, while a negative integer moves backward by {count} candidates.
  - {wrap}? (boolean, default: true) Whether to loop around file or not. Similar to 'wrapscan'.

Lua module: vim.lsp.linked_editing_range lsp-linked_editing_range

The vim.lsp.linked_editing_range module enables "linked editing" via a language server's textDocument/linkedEditingRange request. Linked editing ranges are synchronized text regions, meaning changes in one range are mirrored in all the others. This is helpful in HTML files for example, where the language server can update the text of a closing tag if its opening tag was changed.

LSP spec: https://microsoft.github.io/language-server-protocol/specification/#textDocument_linkedEditingRange

enable({enable}, {filter}) vim.lsp.linked_editing_range.enable()
Enable or disable a linked editing session for the {filter}ed scope. The following is a practical usage example:

```
vim.lsp.start({
  name = 'html',
  cmd = '…',
  on_attach = function(client)
    vim.lsp.linked_editing_range.enable(true, { client_id = client.id })
  end,
})
```

Parameters:
- {enable} (boolean?) true or nil to enable, false to disable.
- {filter} (table?) Optional filters kwargs,
  - {bufnr}? (integer, default: all) Buffer number, or 0 for current buffer, or nil for all.
  - {client_id}? (integer, default: all) Client ID, or nil for all.

Lua module: vim.lsp.log lsp-log

The vim.lsp.log module provides logging for the Nvim LSP client.

When debugging language servers, it is helpful to enable extra-verbose logging of the LSP client RPC events. Example:

```
vim.lsp.log.set_level 'trace'
```

Then try to run the language server, and open the log with:

```
:log lsp
```

Note:
- Remember to DISABLE verbose logging ("debug" or "trace" level), else you may encounter performance issues.
- "ERROR" messages containing "stderr" only indicate that the log was sent to stderr. Many servers send harmless messages via stderr.

get_filename() vim.lsp.log.get_filename()
Returns the log filename.

Return:
(string) log filename

get_level() vim.lsp.log.get_level()
Gets the current log level.

Return:
(integer) current log level

set_format_func({handle}) vim.lsp.log.set_format_func()
Sets the formatting function used to format logs. If the formatting function returns nil, the entry won't be written to the log file.

Parameters:
- {handle} (fun(level:string, ...): string?) Function to apply to log entries. The default will log the level, date, source and line number of the caller, followed by the arguments.

set_level({level}) vim.lsp.log.set_level()
Sets the current log level.

Parameters:
- {level} (string|integer) One of vim.log.levels

Lua module: vim.lsp.on_type_formatting lsp-on_type_formatting

enable({enable}, {filter}) vim.lsp.on_type_formatting.enable()
Enables/disables on-type formatting globally or for the {filter}ed scope. The following are some practical usage examples:

```
-- Enable for all clients
vim.lsp.on_type_formatting.enable()

-- Enable for a specific client
vim.api.nvim_create_autocmd('LspAttach', {
  callback = function(ev)
    local client_id = ev.data.client_id
    local client = assert(vim.lsp.get_client_by_id(client_id))
    if client.name == 'rust-analyzer' then
      vim.lsp.on_type_formatting.enable(true, { client_id = client_id })
    end
  end,
})
```

Parameters:
- {enable} (boolean?) true/nil to enable, false to disable.
- {filter} (table?) Optional filters kwargs:
  - {client_id} (integer?) Client ID, or nil for all.

Lua module: vim.lsp.rpc

lsp-rpc

vim.lsp.rpc.Client Client RPC object

Fields:
- {is_closing} (fun(): boolean) Indicates if the RPC is closing.
- {notify} (fun(method: string, params: any): boolean) Sends a notification to the LSP server.
- {request} (fun(method: string, params: table?, callback: fun(err?: lsp.ResponseError, result: any, request_id: integer), notify_reply_callback?: fun(message_id: integer)):boolean,integer?) Sends a request to the LSP server.
- {terminate} (fun()) Terminates the RPC client.

connect({host_or_path}, {port}) vim.lsp.rpc.connect()
Create a LSP RPC client factory that connects to either:
- a named pipe (windows)
- a domain socket (unix)
- a host and port via TCP

Return a function that can be passed to the cmd field for vim.lsp.start().

Parameters:
- {host_or_path} (string) host to connect to or path to a pipe/domain socket
- {port} (integer?) TCP port to connect to. If absent the first argument must be a pipe

Return:
(fun(dispatchers: vim.lsp.rpc.Dispatchers): vim.lsp.rpc.Client)

format_rpc_error({err}) vim.lsp.rpc.format_rpc_error()
Constructs an error message from an LSP error object.

Parameters:
- {err} (table) The error object

Return:
(string) error_message The formatted error message

vim.lsp.rpc.rpc_response_error()
rpc_response_error({code}, {message}, {data}) Creates an RPC response table error to be sent to the LSP response.

Parameters:
- {code} (integer) RPC error code defined, see vim.lsp.protocol.ErrorCodes
- {message} (string?) arbitrary message to send to server
- {data} (any?) arbitrary data to send to server

Return:
(lsp.ResponseError)

See also:
- lsp.ErrorCodes See vim.lsp.protocol.ErrorCodes

start({cmd}, {dispatchers}, {extra_spawn_params}) vim.lsp.rpc.start() Starts an LSP server process and create an LSP RPC client object to interact with it. Communication with the spawned process happens via stdio. For communication via TCP, spawn a process manually and use vim.lsp.rpc.connect()

Parameters:
- {cmd} (string[]) Command to start the LSP server.
- {dispatchers} (table?) Dispatchers for LSP message types.
  - {notification} (fun(method: string, params: table))
  - {on_error} (fun(code: integer, err: any))
  - {on_exit} (fun(code: integer, signal: integer))
  - {server_request} (fun(method: string, params: table): any?, lsp.ResponseError?)
- {extra_spawn_params} (table?) Additional context for the LSP server process.
  - {cwd}? (string) Working directory for the LSP server process
  - {detached}? (boolean) Detach the LSP server process from the current process
  - {env}? (table<string,string>) Additional environment variables for LSP server process. See vim.system()

Return:
(vim.lsp.rpc.Client)

Lua module: vim.lsp.semantic_tokens

lsp-semantic_tokens

enable({enable}, {filter}) vim.lsp.semantic_tokens.enable()
Enables or disables semantic tokens for the {filter}ed scope.

To "toggle", pass the inverse of is_enabled():

```
vim.lsp.semantic_tokens.enable(not vim.lsp.semantic_tokens.is_enabled())
```

Parameters:
- {enable} (boolean?) true/nil to enable, false to disable
- {filter} (table?) Optional filters kwargs,
  - {bufnr}? (integer, default: all) Buffer number, or 0 for current buffer, or nil for all.
  - {client_id}? (integer, default: all) Client ID, or nil for all.

force_refresh({bufnr}) vim.lsp.semantic_tokens.force_refresh()
Force a refresh of all semantic tokens

Only has an effect if the buffer is currently active for semantic token highlighting (vim.lsp.semantic_tokens.enable() has been called for it)

Parameters:
- {bufnr} (integer?) filter by buffer. All buffers if nil, current buffer if 0

vim.lsp.semantic_tokens.get_at_pos()
get_at_pos({bufnr}, {row}, {col}) Return the semantic token(s) at the given position. If called without arguments, returns the token under the cursor.

Parameters:
- {bufnr} (integer?) Buffer number (0 for current buffer, default)
- {row} (integer?) Position row (default cursor position)
- {col} (integer?) Position column (default cursor position)

Return:
(table?) List of tokens at position. Each token has the following fields:
- line (integer) line number, 0-based
- start_col (integer) start column, 0-based
- end_line (integer) end line number, 0-based
- end_col (integer) end column, 0-based
- type (string) token type as string, e.g. "variable"
- modifiers (table) token modifiers as a set. E.g., { static = true, readonly = true }
- client_id (integer)

vim.lsp.semantic_tokens.highlight_token()
highlight_token({token}, {bufnr}, {client_id}, {hl_group}, {opts}) Highlight a semantic token.

Apply an extmark with a given highlight group for a semantic token. The mark will be deleted by the semantic token engine when appropriate; for example, when the LSP sends updated tokens. This function is intended for use inside LspTokenUpdate callbacks.

Parameters:
- {token} (table) Semantic token, provided as ev.data.token in LspTokenUpdate
- {bufnr} (integer) Buffer to highlight, or 0 for current buffer.
- {client_id} (integer) ID of the vim.lsp.Client
- {hl_group} (string) Highlight group name
- {opts} (table?) Optional parameters:
  - {priority}? (integer, default: vim.hl.priorities.semantic_tokens + 3) Priority for the applied extmark.

is_enabled({filter}) vim.lsp.semantic_tokens.is_enabled()
Query whether semantic tokens is enabled in the {filter}ed scope

Parameters:
- {filter} (table?) Optional filters kwargs,
  - {bufnr}? (integer, default: all) Buffer number, or 0 for current buffer, or nil for all.
  - {client_id}? (integer, default: all) Client ID, or nil for all.

Lua module: vim.lsp.util

lsp-util

vim.lsp.util.open_floating_preview.Opts

Fields:
- {anchor_bias}? ('auto'|'above'|'below', default: 'auto') Adjusts placement relative to cursor.
  - "auto": place window based on which side of the cursor has more lines
  - "above": place the window above the cursor unless there are not enough lines to display the full window height.
  - "below": place the window below the cursor unless there are not enough lines to display the full window height.
- {border}? ('|'none'|'single'|'double'|'rounded'|'solid'|'shadow'|'bold'|(string|[string,string])[]) override border
- {close_events}? (table) List of events that closes the floating window
- {focus}? (boolean, default: true) If true, and if {focusable} is also true, focus an existing floating window with the same {focus_id}
- {focus_id}? (string) If a popup with this id is opened, then focus it
- {focusable}? (boolean, default: true) Make float focusable.
- {height}? (integer) Height of floating window
- {max_height}? (integer) Maximal height of floating window
- {max_width}? (integer) Maximal width of floating window
- {offset_x}? (integer) offset to add to col
- {offset_y}? (integer) offset to add to row
- {relative}? ('mouse'|'cursor'|'editor') (default: 'cursor')

- {title}? (string|[string,string][])
- {title_pos}? ('left'|'center'|'right')
- {width}? (integer) Width of floating window
- {wrap}? (boolean, default: true) Wrap long lines
- {wrap_at}? (integer) Character to wrap at for computing height when wrap is enabled
- {zindex}? (integer) override zindex, defaults to 50

vim.lsp.util.apply_text_document_edit()

apply_text_document_edit({text_document_edit}, {index}, {position_encoding}, {change_annotations}) Applies a TextDocumentEdit, which is a list of changes to a single document.

Parameters:
- {text_document_edit} (lsp.TextDocumentEdit)
- {index} (integer?) Optional index of the edit, if from a list of edits (or nil, if not from a list)
- {position_encoding} ('utf-8'|'utf-16'|'utf-32')
- {change_annotations} (table<string, lsp.ChangeAnnotation>?)

See also:
- https://microsoft.github.io/language-server-protocol/specifications/specification-current/#textDocumentEdit

vim.lsp.util.apply_text_edits()

apply_text_edits({text_edits}, {bufnr}, {position_encoding}, {change_annotations}, {opts}) Applies a list of text edits to a buffer.
Note: this mutates text_edits (sorts in-place and adds _index fields).

Parameters:
- {text_edits} ((lsp.TextEdit|lsp.AnnotatedTextEdit)[])
- {bufnr} (integer) Buffer id
- {position_encoding} ('utf-8'|'utf-16'|'utf-32')
- {change_annotations} (table<string, lsp.ChangeAnnotation>?)
- {opts} (table?) A table with the following fields:
  - {keep_cursor}? (boolean, default: false) Keep the cursor on the text it is on, instead of letting the edits move it.

See also:
- https://microsoft.github.io/language-server-protocol/specifications/specification-current/#textEdit

vim.lsp.util.apply_workspace_edit()

apply_workspace_edit({workspace_edit}, {position_encoding}) Applies a WorkspaceEdit.

Parameters:
- {workspace_edit} (lsp.WorkspaceEdit)
- {position_encoding} ('utf-8'|'utf-16'|'utf-32') (required)

See also:
- https://microsoft.github.io/language-server-protocol/specifications/specification-current/#workspace_applyEdit

buf_clear_references({bufnr}) vim.lsp.util.buf_clear_references()
Removes document highlights from a buffer.

Parameters:
- {bufnr} (integer?) Buffer id

vim.lsp.util.buf_highlight_references()

buf_highlight_references({bufnr}, {references}, {position_encoding}) Shows a list of document highlights for a certain buffer.

Parameters:
- {bufnr} (integer) Buffer id
- {references} (lsp.DocumentHighlight[]) objects to highlight
- {position_encoding} ('utf-8'|'utf-16'|'utf-32')

See also:
- https://microsoft.github.io/language-server-protocol/specification/#textDocumentContentChangeEvent

vim.lsp.util.convert_input_to_markdown_lines()

convert_input_to_markdown_lines({input}, {contents}) Converts any of MarkedString | MarkedString[] | MarkupContent into a list of lines containing valid markdown. Useful to populate the hover window for textDocument/hover, for parsing the result of textDocument/signatureHelp, and potentially others.

Note that if the input is of type MarkupContent and its kind is plaintext, then the corresponding value is returned without further modifications.

Parameters:
- {input} (lsp.MarkedString|lsp.MarkedString[]|lsp.MarkupContent)
- {contents} (string[]?) List of strings to extend with converted lines. Defaults to {}.

Return:
(string[]) extended with lines of converted markdown.

See also:
- https://microsoft.github.io/language-server-protocol/specifications/specification-current/#textDocument_hover

vim.lsp.util.convert_signature_help_to_markdown_lines()

convert_signature_help_to_markdown_lines({signature_help}, {ft}, {triggers}) Converts textDocument/signatureHelp response to markdown lines.

Parameters:
- {signature_help} (lsp.SignatureHelp) Response of textDocument/SignatureHelp
- {ft} (string?) filetype that will be use as the lang for the label markdown code block
- {triggers} (string[]?) list of trigger characters from the lsp server. used to better determine parameter offsets

Return (multiple):
(`string[]?`) lines of converted markdown. (Range4?) highlight range for the active parameter

See also:
- https://microsoft.github.io/language-server-protocol/specifications/specification-current/#textDocument_signatureHelp

get_effective_tabstop({`bufnr`}) vim.lsp.util.get_effective_tabstop()
Returns indentation size.

Parameters:
- {`bufnr`} (`integer?`) Buffer handle, defaults to current

Return:
(`integer`) indentation size

See also:
- 'shiftwidth'

vim.lsp.util.locations_to_items()
locations_to_items({`locations`}, {`position_encoding`}) Returns the items with the byte position calculated correctly and in sorted order, for display in quickfix and location lists.

The `user_data` field of each resulting item will contain the original `Location` or `LocationLink` it was computed from.

The result can be passed to the {`list`} argument of setqflist() or setloclist().

Parameters:
- {`locations`} (`lsp.Location[]|lsp.LocationLink[]`)
- {`position_encoding`} (`'utf-8'|'utf-16'|'utf-32'`)

Return:
(`vim.quickfix.entry[]`) See setqflist() for the format

vim.lsp.util.make_floating_popup_options()
make_floating_popup_options({`width`}, {`height`}, {`opts`}) Creates a table with sensible default options for a floating window. The table can be passed to nvim_open_win().

Parameters:
- {`width`} (`integer`) window width (in character cells)
- {`height`} (`integer`) window height (in character cells)
- {`opts`} (`vim.lsp.util.open_floating_preview.Opts?`)

Return:
(`vim.api.keyset.win_config`)

vim.lsp.util.make_formatting_params()
make_formatting_params({`options`}) Creates a `DocumentFormattingParams` object for the current buffer and cursor position.

Parameters:
- {`options`} (`lsp.FormattingOptions?`) with valid `FormattingOptions` entries

Return:
(`lsp.DocumentFormattingParams`) object

See also:
- https://microsoft.github.io/language-server-protocol/specifications/specification-current/#textDocument_formatting

vim.lsp.util.make_given_range_params()
make_given_range_params({`start_pos`}, {`end_pos`}, {`bufnr`}, {`position_encoding`}) Using the given range in the current buffer, creates an object that is similar to vim.lsp.util.make_range_params().

Parameters:
- {`start_pos`} (`[integer,integer]?`) {`row,col`} mark-indexed position. Defaults to the start of the last visual selection.
- {`end_pos`} (`[integer,integer]?`) {`row,col`} mark-indexed position. Defaults to the end of the last visual selection.
- {`bufnr`} (`integer?`) buffer handle or 0 for current, defaults to current
- {`position_encoding`} (`'utf-8'|'utf-16'|'utf-32'`)

Return:
(`{ textDocument: { uri: lsp.DocumentUri }, range: lsp.Range }`)

vim.lsp.util.make_position_params()
make_position_params({`win`}, {`position_encoding`}) Creates a `TextDocumentPositionParams` object for the current buffer and cursor position.

Parameters:
- {`win`} (`integer?`) window-ID or 0 for current, defaults to current
- {`position_encoding`} (`'utf-8'|'utf-16'|'utf-32'`)

Return:
(`lsp.TextDocumentPositionParams`)

See also:
- https://microsoft.github.io/language-server-protocol/specifications/specification-current/#textDocumentPositionParams

vim.lsp.util.make_range_params()
make_range_params({`win`}, {`position_encoding`}) Using the current position in the current buffer, creates an object that can be used as a building block for several LSP requests, such as `textDocument/codeAction`, `textDocument/colorPresentation`, `textDocument/rangeFormatting`.

Parameters:
- {`win`} (`integer?`) window-ID or 0 for current, defaults to current
- {`position_encoding`} (`"utf-8"|"utf-16"|"utf-32"`)

Return:
(`{ textDocument: { uri: lsp.DocumentUri }, range: lsp.Range }`)

vim.lsp.util.make_text_document_params()

make_text_document_params({`bufnr`}) Creates a `TextDocumentIdentifier` object for the current buffer.

Parameters:
- {`bufnr`} (`integer?`) Buffer handle, defaults to current

Return:
(`lsp.TextDocumentIdentifier`)

See also:
- https://microsoft.github.io/language-server-protocol/specifications/specification-current/#textDocumentIdentifier

vim.lsp.util.make_workspace_params()

make_workspace_params({`added`}, {`removed`}) Create the workspace params

Parameters:
- {`added`} (`lsp.WorkspaceFolder[]`)
- {`removed`} (`lsp.WorkspaceFolder[]`)

Return:
(`lsp.DidChangeWorkspaceFoldersParams`)

vim.lsp.util.open_floating_preview()

open_floating_preview({`contents`}, {`syntax`}, {`opts`}) Shows contents in a floating window.

Parameters:
- {`contents`} (`table`) of lines to show in window
- {`syntax`} (`string`) of syntax to set for opened buffer
- {`opts`} (`vim.lsp.util.open_floating_preview.Opts?`) with optional fields (additional keys are filtered with vim.lsp.util.make_floating_popup_options() before they are passed on to nvim_open_win())

Return (multiple):
(`integer`) bufnr of newly created float window (`integer`) winid of newly created float window preview window

preview_location({`location`}, {`opts`}) vim.lsp.util.preview_location()
Previews a location in a floating window

behavior depends on type of location:
- for Location, range is shown (e.g., function definition)
- for LocationLink, targetRange is shown (e.g., body of function definition)

Parameters:
- {`location`} (`lsp.Location|lsp.LocationLink`)
- {`opts`} (`vim.lsp.util.open_floating_preview.Opts?`)

Return (multiple):
(`integer?`) buffer id of float window (`integer?`) window id of float window

rename({`old_fname`}, {`new_fname`}, {`opts`}) vim.lsp.util.rename()
Rename old_fname to new_fname

Existing buffers are renamed as well, while maintaining their bufnr.

It deletes existing buffers that conflict with the renamed file name only when
- `opts` requests overwriting; or
- the conflicting buffers are not loaded, so that deleting them does not result in data loss.

Parameters:
- {`old_fname`} (`string`)
- {`new_fname`} (`string`)
- {`opts`} (`table?`) Options:
  - {`ignoreIfExists`}? (`boolean`)
  - {`overwrite`}? (`boolean`)

vim.lsp.util.show_document()

show_document({`location`}, {`position_encoding`}, {`opts`}) Shows document and optionally jumps to the location.

Parameters:
- {`location`} (`lsp.Location|lsp.LocationLink`)
- {`position_encoding`} (`'utf-8'|'utf-16'|'utf-32'`)
- {`opts`} (`table?`) A table with the following fields:
  - {`focus`}? (`boolean`) Whether to focus/jump to location if possible. (defaults: true)
  - {`reuse_win`}? (`boolean`) Jump to existing window if buffer is already open.

Return:
(`boolean`) `true` if succeeded

vim.lsp.util.symbols_to_items()

symbols_to_items({`symbols`}, {`bufnr`}, {`position_encoding`}) Converts symbols to quickfix list items.

Parameters:
- {`symbols`} (`lsp.DocumentSymbol[]|lsp.SymbolInformation[]|lsp.WorkspaceSymbol[]`) list of symbols
- {`bufnr`} (`integer?`) buffer handle or 0 for current, defaults to current
- {`position_encoding`} (`'utf-8'|'utf-16'|'utf-32'`)

Return:
(`vim.quickfix.entry[]`) See setqflist() for the format

Lua module: vim.lsp.protocol

lsp-protocol

vim.lsp.protocol.make_client_capabilities()

make_client_capabilities() Gets a new ClientCapabilities object describing the LSP client capabilities.

Return:
(`lsp.ClientCapabilities`)

vim.lsp.protocol.resolve_capabilities()

resolve_capabilities({`server_capabilities`}) Creates a normalized object describing LSP server capabilities.

Parameters:
- {`server_capabilities`} (`table`) Table of capabilities supported by the server

Return:
(`lsp.ServerCapabilities?`) Normalized table of capabilities

Structured async API for Lua code that waits on event-loop work.

vim.async lets Lua code wait for timers, callbacks, and other tasks without blocking Nvim's event loop. Async work runs inside tasks, which can pause at checkpoints and manage child tasks created while they are running.

Start async work with vim.async.run(). Inside a task, use vim.async.await() to wait for callback-style APIs or other tasks without blocking the event loop. Use vim.async.pawait() when an awaited operation can fail and the current task should continue.

Examples in this help use local async = vim.async for brevity.

Example: run async work without blocking Nvim:

```
local async = vim.async

async.run(function()
  vim.notify('waiting...')
  async.sleep(1000)
  vim.notify('done')
end)
```

Example: await a callback-style API. Callback results are returned unchanged, so an error-first callback still returns err, value:

```
local async = vim.async

async.run(function()
  local err, stat = async.await(2, fs_stat, 'notes.txt')
  if err then
    error(err, 0)
  end
  print(('notes.txt is %d bytes'):format(stat.size))
end)
```

A task has two roles:

- it is a handle that can be awaited, waited for, or closed
- it is a scope for child tasks created while the task is running

vim.async.run() creates a task. A top-level task starts immediately. A task created while another task is running becomes a child of that task, and its function starts when the parent reaches its next checkpoint. A parent task finishes only after its attached children finish. If a child fails without being handled, the parent fails and closes the remaining children.

Use Task:detach() for background work that should keep running after the current task finishes. A detached task becomes top-level work; the original parent no longer waits for it or closes it.

Awaiting a task observes that task's result; it does not attach the task to the awaiter or change which task owns it. Ownership is decided when the task is created.

Scheduling is cooperative. When a task awaits a timer, I/O operation, callback, or another task, vim.async saves the Lua stack and returns control to the event loop. Other callbacks can run while the task is paused. Nothing interrupts synchronous Lua code in the middle of a stack frame.

Checkpoints are the places where a task can pause, start pending child tasks, observe cancellation, and receive unhandled child failures. Inside a task, these operations are checkpoints:

- vim.async.await(...)
- vim.async.pawait(...)
- vim.async.checkpoint()
- successful return from the task function, which is the final checkpoint for child management

Convenience APIs such as vim.async.sleep() and vim.async.timeout() can also checkpoint because they call checkpointing APIs internally.

Closing a task closes its attached children. Cancellation is cooperative: Task:close() marks a task as closing, and the task observes that state at a checkpoint. If a task is suspended on a closable operation such as a timer or child task, vim.async closes that operation before reporting the cancellation.

Use vim.async.await() inside a task to suspend until work completes. It accepts a task, a callback-taking function, or an argument position plus a callback-taking function. await(task) returns the task result or raises the task failure. vim.async.pawait() is the async counterpart to pcall() for recoverable awaited-operation failures; it returns ok, ... instead of failing the current task for that awaited operation. It does not suppress cancellation or a failure already pending on the current task.

From synchronous code, use Task:wait() or Task:pwait() to pump the event loop until the task completes. Use Task:on_complete() to observe completion without blocking.

Coordination helpers work with task handles. vim.async.iter() yields completed task handles in completion order, vim.async.timeout() awaits a task with a deadline, and vim.async.semaphore(permits) creates a vim.async.Semaphore that limits how many tasks can hold a permit for a section at once.

vim.async.Closable

Fields:

- {close} (fun(self, callback?: fun()))
- {is_closing}? (fun(self): boolean)

vim.async.Semaphore

A semaphore manages an internal permit counter. Semaphore:acquire() consumes one permit and Semaphore:release() returns one permit. If no permits are available, acquire() suspends the current task until another task releases one.

The preferred way to use a Semaphore is with the with() method, which automatically acquires and releases the semaphore around a function call. This is useful for limiting sections that start external work and then await it, such as file reads, requests, or subprocesses.

```
local async = vim.async

async.run(function()
  local limit = async.semaphore(4)
  local tasks = {}

  for _, path in ipairs(paths) do
    table.insert(tasks, async.run(function()
      return limit:with(function()
        return read_file(path)
      end)
    end))
  end

  local next_task = async.iter(tasks)
  while true do
    local task = next_task()
    if task == nil then
      break
    end
    async.await(task)
  end
end)
```

Fields:

- {acquire} (fun(self: vim.async.Semaphore)) See Semaphore:acquire().
- {release} (fun(self: vim.async.Semaphore)) See Semaphore:release().
- {with} (fun(self: vim.async.Semaphore, fn: async fun(): R...): R...) See Semaphore:with().

vim.async.Task

Extends: vim.async.Closable

A coroutine-backed async operation and concurrency scope.

Use vim.async.run() to create tasks. A task may be awaited by more than one waiter. When a task is created inside another task, it is attached to that parent and becomes part of the parent's concurrency scope.

Fields:

- {close} (fun(self: vim.async.Task, callback: fun()?)) See Task:close().
- {completed} (fun(self: vim.async.Task): boolean) See Task:completed().
- {detach} (fun(self: vim.async.Task): vim.async.Task<R>) See Task:detach().
- {name}? (string) Name of the task

- {on_complete} (fun(self: vim.async.Task, callback: fun(err?: any, ...: R...)): fun())

  See Task:on_complete().
- {pwait} (fun(self: vim.async.Task, timeout: integer?): boolean, R...)

  See Task:pwait().
- {raise_on_error} (fun(self: vim.async.Task): vim.async.Task<R>) See

  Task:raise_on_error().
- {status} (fun(self: vim.async.Task): "running"|"awaiting"|"normal"|"completed")

  See Task:status().
- {traceback} (fun(self: vim.async.Task, msg: string?, level: integer?): string)

  See Task:traceback().
- {wait} (fun(self: vim.async.Task, timeout: integer?): R...)

  See Task:wait().

await({...}) vim.async.await()

Suspend the current task until an awaitable completes.

Accepts a task, a callback-taking function, or an argument position plus a callback-taking function. Raises awaited errors and current task-control errors.

The callback forms return callback arguments unchanged:

```
local async = vim.async

async.run(function()
  local err, stat = async.await(2, vim.uv.fs_stat, 'notes.txt')
  if err then
    error(err, 0)
  end
  print(stat.size)
end)
```

If a callback API starts cancellable work, return a closable handle from the await callback. Task:close() will close that handle if cancellation arrives while the task is suspended there.

```
local async = vim.async

async.run(function()
  local lines = async.await(function(done)
    return start_read_lines('notes.txt', done)
  end)
  render(lines)
end)
```

Parameters:
- {...} (any) see overloads

Overloads:
- async fun(func: (fun(callback: fun(...: R...)): vim.async.Closable?)): R...
- async fun(argc: integer, func: (fun(...: T..., callback: fun(...: R...)): vim.async.Closable?), ...: T...): R...
- async fun(task: vim.async.Task<R>): R...

Return:
(R...)

checkpoint() vim.async.checkpoint()

Start pending child tasks and deliver pending cancellation or task failure from the current task.

This does not yield to the event loop.

Use this after cleanup code that catches an async failure or close signal, so persistent task state is delivered again before normal execution continues.

```
local ok, err = pcall(cleanup_sensitive_work)
cleanup_resources()
vim.async.checkpoint()
if not ok then
  error(err, 0)
end
```

is_closing() vim.async.is_closing()

Returns true if the current task has been closed.

Can be used in an async function to do cleanup when a task is closing.

```
while not vim.async.is_closing() do
  poll_once()
  vim.async.sleep(1000)
end
```

Return:
(boolean)

iter({tasks}) vim.async.iter()

Iterate completed tasks in completion order.

The iterator yields task handles, not task results. Use vim.async.await() or vim.async.pawait() to retrieve each result. The tasks are observed in the order they complete, regardless of the order in the input list.

```
local async = vim.async

async.run(function()
  local tasks = {
    async.run(function() return 'cache', read_cache() end):detach(),
    async.run(function() return 'disk', read_file() end):detach(),
  }

  for task in async.iter(tasks) do
    local ok, source, text = async.pawait(task)
    if ok then
      for _, other in ipairs(tasks) do
        if other ~= task then
          other:close()
        end
      end
      print(('loaded from %s'):format(source))
      return text
    end
  end
end)
```

If code must support PUC Lua 5.1, use the direct-call form instead of a generic for loop. The iterator may need to suspend while waiting for the next completed task, and PUC Lua 5.1 cannot yield from a generic-for iterator call.

```
local next_task = async.iter(tasks)
while true do
  local task = next_task()
  if task == nil then
    break
  end
  async.await(task)
end
```

Parameters:
- {tasks} (vim.async.Task<R>[]) A list of tasks to wait for and iterate over.

Return:
(async fun(): vim.async.Task<R>?) iterator that yields each completed task.

pawait({...}) vim.async.pawait()

Protected await.

Async counterpart to pcall(). Accepts the same forms as vim.async.await(), but returns a leading ok boolean for awaited-operation failures.

Use this when the awaited task or operation is allowed to fail and the current task should continue. Cancellation or already pending failure from the current task is not protected.

```
local async = vim.async

async.run(function()
  local ok, text_or_err = async.pawait(async.run(read_file, 'notes.txt'))
  if not ok then
    text_or_err = ''
  end

  show_buffer(text_or_err)
end)
```

Parameters:

- {...} (any) see overloads

Overloads:
- async fun(func: (fun(callback: fun(...: R...)): vim.async.Closable?)): boolean, R...

- async fun(argc: integer, func: (fun(...: T..., callback: fun(...: R...)): vim.async.Closable?), ...: T...): boolean, R...

- async fun(task: vim.async.Task<R>): boolean, R...

Return (multiple):
(boolean) ok
(R...) ... result or error
(_overload) true
(R...)
(_overload) false
(any)

run({func}, {...}) vim.async.run()

Create a task from an async function.

Top-level tasks start immediately. Child tasks are attached immediately and first run when their parent reaches a checkpoint.

Creating a task decides ownership. Awaiting the task later only observes its result; it does not attach the task to the awaiter.

```
local async = vim.async

async.run(function()
  local child = async.run(function()
    return read_file('notes.txt')
  end)

  local text = async.await(child)
  show_buffer(text)
end)
```

A task created from synchronous code is top-level:

```
local task = vim.async.run(function()
  vim.async.sleep(100)
  return 'done'
end)

print(task:wait())
```

Parameters:
- {func} (async fun(...: T...): R...)

- {...} (T...) Arguments to pass to the function

Overloads:
- fun(name: string, func: async fun(...: T...), ...: T...): vim.async.Task<R...>

Return:
(vim.async.Task<R...>)

semaphore({permits}) vim.async.semaphore()

Create an async semaphore that allows up to a given number of acquisitions.

Prefer Semaphore:with() for most uses so permits are released reliably. Use Semaphore:acquire() and Semaphore:release() directly only when the acquire and release points cannot be expressed as one function call.

Parameters:
- {permits} (integer?) (default: 1)

Return:
(vim.async.Semaphore)

Semaphore:acquire() Semaphore:acquire()

Acquire a semaphore permit.

If the internal counter is greater than zero, decrement it by 1 and return immediately. If it is 0, wait until Semaphore:release() is called.

Semaphore:release() Semaphore:release()

Release a semaphore permit.

Increments the internal counter by 1 and can wake a task waiting in Semaphore:acquire().

Calling this more times than permits were acquired raises an error.

Semaphore:with({fn}) Semaphore:with()

Executes a function while holding one semaphore permit.

This acquires the semaphore before running the function and releases it after the function completes, even if it errors or the current task is closed.

Parameters:
- {fn} (async fun(): R...) Function to execute within the semaphore's context.

Return:
(R...) Result(s) of the executed function.

sleep({duration}) vim.async.sleep()

Asynchronously sleep for a given duration.

Suspends the current task for the given duration without blocking the event loop. After the delay and timer cleanup complete, sleep() returns to its caller through the runtime's schedule hook.

```
vim.async.run(function()
  vim.async.sleep(100)
  vim.notify('resumed later')
end)
```

Parameters:
- {duration} (integer) ms

Task:close({callback}) Task:close()

Request cooperative close for the task and all of its children.

The optional callback observes task completion and may run immediately if the task has already completed.

Closing is cooperative. The task observes the close request at a checkpoint such as vim.async.await() or vim.async.checkpoint(). If the task is suspended on an owned closable operation, that operation is closed before the task reports "closed".

Parameters:
- {callback} (fun()?)

Task:completed() Task:completed()

Returns whether the Task has completed.

Return:
(boolean)

Task:detach() Task:detach()

Detach a task from its parent.

The task becomes a top-level task. If it was waiting for a parent checkpoint, it is scheduled to start.

Use this for background work that should not be cancelled when the current task finishes. Detached task failures no longer fail the original parent, so observe them explicitly with Task:on_complete(), Task:wait(), or vim.async.await().

```
vim.async.run(function()
  while true do
    refresh_index()
    vim.async.sleep(1000)
  end
end):detach()
```

Return:
(vim.async.Task<R>)

Task:on_complete({callback}) Task:on_complete()

Add a callback to be run when the Task has completed.

If the Task is already done when this method is called, the callback is called immediately with the results.

This only observes completion. It does not start a pending task.

Parameters:
- {callback} (fun(err?: any, ...: R...))

Return:
(fun()) unsubscribe

Task:pwait({timeout}) Task:pwait()

Protected-call version of Task:wait().

Equivalent to pcall(task.wait, task, timeout).

```
local ok, result_or_err = task:pwait(1000)
if not ok then
  vim.notify(tostring(result_or_err), vim.log.levels.ERROR)
end
```

Parameters:
- {timeout} (integer?)

Return (multiple):
(boolean)
(R...)

Task:raise_on_error() Task:raise_on_error()

Raise this task's error when it completes.

Use this for detached or top-level fire-and-forget tasks whose completion will not otherwise be observed. Attached task errors already propagate to their parent.

Detached tasks do not raise errors automatically. Detaching changes ownership only; their completion can still be handled with vim.async.await(), Task:wait(), Task:pwait(), or Task:on_complete().

Return:
(vim.async.Task<R>) self

Task:status() Task:status()

Returns the status of the task:
- "running": task is currently executing Lua code
- "normal": task is active but another coroutine is running
- "awaiting": task is suspended at a checkpoint or waiting for children
- "completed": task and all attached children have completed

Return:
("running"|"awaiting"|"normal"|"completed")

Task:traceback({msg}, {level}) Task:traceback()

Get the traceback of a task when it is not active. Will also get the traceback of nested tasks.

Parameters:
- {msg} (string?)
- {level} (integer?)

Return:
(string) traceback

Task:wait({timeout}) Task:wait()

Synchronously wait for the Task to complete.

If a timeout is provided, waits for the given time in milliseconds before failing with "timeout". With no timeout, waits indefinitely.

This is for synchronous code. Inside a task, prefer vim.async.await() so the current task suspends instead of pumping the event loop itself.

```
local result = task:wait(10) -- wait for 10ms or raise "timeout"

local result = task:wait() -- wait indefinitely
```

Parameters:
- {timeout} (integer?)

Return:
(R...)

timeout({duration}, {task}) vim.async.timeout()

Await a task with a timeout.

If the task completes first, returns or raises the task result as vim.async.await() would. If the deadline wins, closes the task and

raises "timeout" after the target task finishes cancellation cleanup.

```
local async = vim.async

async.run(function()
  local task = async.run(read_file, 'notes.txt')
  local text = async.timeout(5000, task)
  show_buffer(text)
end)
```

Parameters:

- {duration} (integer) Timeout duration in milliseconds
- {task} (vim.async.Task<R>)

Return:

(R...)

wrap({argc}, {func}) vim.async.wrap()

Create an async function from a callback-style function.

The callback is inserted at argument position argc. If func returns a closable handle, it is closed when the awaiting task is cancelled.

This is a reusable wrapper around vim.async.await(). Use it when the same callback API is awaited from more than one place.

```
local async = vim.async
local fs_stat = async.wrap(2, vim.uv.fs_stat)

async.run(function()
  local err, stat = fs_stat(vim.api.nvim_buf_get_name(0))
  if not err and stat then
    print(stat.size)
  end
end)
```

Parameters:

- {argc} (integer)
- {func} (fun(...: T..., callback: fun(...: R...)): vim.async.Closable?)

Return:

(async fun(...: T...): R...)

Lua BitOp

Adapted from https://bitop.luajit.org.

Lua BitOp is a C extension module for Lua 5.1/5.2 which adds bitwise operations on numbers.

API FUNCTIONS

lua-bit-api

This list of API functions is not intended to replace a tutorial. If you are not familiar with the terms used, you may want to study the Wikipedia article on bitwise operations (https://en.wikipedia.org/wiki/Bitwise_operation) first.

Loading the BitOp module

lua-bit-module

The suggested way to use the BitOp module is to add the following to the start of every Lua file that needs one of its functions:

```
local bit = require("bit")
```

This makes the dependency explicit, limits the scope to the current file and provides faster access to the bit.* functions, too. It's good programming practice not to rely on the global variable bit being set (assuming some other part of your application has already loaded the module). The require function ensures the module is only loaded once, in any case.

Defining Shortcuts

lua-bit-shortcuts

It's a common (but not a required) practice to cache often used module functions in locals. This serves as a shortcut to save some typing and also speeds up resolving them (only relevant if called hundreds of thousands of times).

```
local bnot = bit.bnot
local band, bor, bxor = bit.band, bit.bor, bit.bxor
local lshift, rshift, rol = bit.lshift, bit.rshift, bit.rol
-- etc...

-- Example use of the shortcuts:
local function tr_i(a, b, c, d, x, s)
  return rol(bxor(c, bor(b, bnot(d))) + a + x, s) + b
end
```

Remember that `and`, `or` and `not` are reserved keywords in Lua. They cannot be used for variable names or literal field names. That's why the corresponding bitwise functions have been named `band`, `bor`, and `bnot` (and `bxor` for consistency).

While we are at it: a common pitfall is to use bit as the name of a local temporary variable — well, don't! :-)

About the Examples

The examples below show small Lua one-liners. Their expected output is shown after `-->`. This is interpreted as a comment marker by Lua so you can cut & paste the whole line to a Lua prompt and experiment with it.

Note that all bit operations return signed 32 bit numbers (rationale). And these print as signed decimal numbers by default.

For clarity the examples assume the definition of a helper function `printx()`. This prints its argument as an unsigned 32 bit hexadecimal number on all platforms:

```
function printx(x)
  print("0x"..bit.tohex(x))
end
```

Bit operations

lua-bitop

bit.tobit({x}) bit.tobit()

Normalizes a number to the numeric range for bit operations and returns it. This function is usually not needed since all bit operations already normalize all of their input arguments. See lua-bit-semantics.

Example:

```
print(0xffffffff)               --> 4294967295 (see Note)
print(bit.tobit(0xffffffff))    --> -1
printx(bit.tobit(0xffffffff))   --> 0xffffffff
print(bit.tobit(0xffffffff + 1)) --> 0
print(bit.tobit(2^40 + 1234))   --> 1234
```

Note: lua-bit-hex-literals explains why the numbers printed in the first two lines differ (if your Lua installation uses a double number type).

bit.tohex({x} [, {n}]) bit.tohex()

Converts its first argument to a hex string. The number of hex digits is given by the absolute value of the optional second argument. Positive numbers between 1 and 8 generate lowercase hex digits. Negative numbers generate uppercase hex digits. Only the least-significant 4*|n| bits are used. The default is to generate 8 lowercase hex digits.

Example:

```
print(bit.tohex(1))              --> 00000001
print(bit.tohex(-1))             --> ffffffff
print(bit.tohex(0xffffffff))     --> ffffffff
print(bit.tohex(-1, -8))         --> FFFFFFFF
print(bit.tohex(0x21, 4))        --> 0021
print(bit.tohex(0x87654321, 4))  --> 4321
```

bit.bnot({x}) bit.bnot()

Returns the bitwise `not` of its argument.

Example:

```
print(bit.bnot(0))            --> -1
printx(bit.bnot(0))           --> 0xffffffff
print(bit.bnot(-1))           --> 0
print(bit.bnot(0xffffffff))   --> 0
printx(bit.bnot(0x12345678))  --> 0xedcba987
```

bit.bor({...}) bit.bor()
bit.band({...}) bit.band()
bit.bxor({...}) bit.bxor()

Returns either the bitwise `or`, bitwise and, or bitwise `xor` of all of its arguments. Note that more than two arguments are allowed.

Example:

```
print(bit.bor(1, 2, 4, 8))               --> 15
printx(bit.band(0x12345678, 0xff))       --> 0x00000078
printx(bit.bxor(0xa5a5f0f0, 0xaa55ff00))  --> 0x0ff00ff0
```

bit.lshift({x}, {n}) bit.lshift()
bit.rshift{(x}, {n}) bit.rshift()
bit.arshift{(x}, {n}) bit.arshift()

Returns either the bitwise `logical left-shift`, bitwise `logical right-shift`, or bitwise `arithmetic right-shift` of its first argument by the number of bits given by the second argument.

Logical shifts treat the first argument as an unsigned number and shift in 0-bits. Arithmetic right-shift treats the most-significant bit as a sign bit and replicates it. Only the lower 5 bits of the shift count are used (reduces to the range [0..31]).

Example:

```
print(bit.lshift(1, 0))             --> 1
print(bit.lshift(1, 8))             --> 256
print(bit.lshift(1, 40))            --> 256
print(bit.rshift(256, 8))           --> 1
print(bit.rshift(-256, 8))          --> 16777215
print(bit.arshift(256, 8))          --> 1
print(bit.arshift(-256, 8))         --> -1
printx(bit.lshift(0x87654321, 12))  --> 0x54321000
printx(bit.rshift(0x87654321, 12))  --> 0x00087654
printx(bit.arshift(0x87654321, 12))  --> 0xfff87654
```

bit.rol({x}, {n}) bit.rol()
bit.ror({x}, {n}) bit.ror()

Returns either the bitwise `left rotation`, or bitwise `right rotation` of its first argument by the number of bits given by the second argument. Bits shifted out on one side are shifted back in on the other side.

Only the lower 5 bits of the rotate count are used (reduces to the range [0..31]).

Example:

```
printx(bit.rol(0x12345678, 12))  --> 0x45678123
printx(bit.ror(0x12345678, 12))  --> 0x67812345
```

bit.bswap({x}) bit.bswap()

Swaps the bytes of its argument and returns it. This can be used to convert little-endian 32 bit numbers to big-endian 32 bit numbers or vice versa.

Example:

```
printx(bit.bswap(0x12345678)) --> 0x78563412
printx(bit.bswap(0x78563412)) --> 0x12345678
```

Example Program

This is an implementation of the (naïve) Sieve of Eratosthenes algorithm. It counts the number of primes up to some maximum number.

A Lua table is used to hold a bit-vector. Every array index has 32 bits of the vector. Bitwise operations are used to access and modify them. Note that the shift counts don't need to be masked since this is already done by the BitOp shift and rotate functions.

```
local bit = require("bit")
local band, bxor = bit.band, bit.bxor
local rshift, rol = bit.rshift, bit.rol

local m = tonumber(arg and arg[1]) or 100000
if m < 2 then m = 2 end
local count = 0
local p = {}

for i=0,(m+31)/32 do p[i] = -1 end

for i=2,m do
  if band(rshift(p[rshift(i, 5)], i), 1) ~= 0 then
    count = count + 1
    for j=i+i,m,i do
      local jx = rshift(j, 5)
      p[jx] = band(p[jx], rol(-2, j))
    end
  end
end

io.write(string.format("Found %d primes up to %d\n", count, m))
```

Lua BitOp is quite fast. This program runs in less than 90 milliseconds on a 3 GHz CPU with a standard Lua installation, but performs more than a million calls to bitwise functions. If you're looking for even more speed, check out lua-luajit.

Caveats

lua-bit-caveats

Signed Results

Returning signed numbers from bitwise operations may be surprising to programmers coming from other programming languages which have both signed and unsigned types. But as long as you treat the results of bitwise operations uniformly everywhere, this shouldn't cause any problems.

Preferably format results with `bit.tohex` if you want a reliable unsigned string representation. Avoid the `"%x"` or `"%u"` formats for `string.format`. They fail on some architectures for negative numbers and can return more than 8 hex digits on others.

You may also want to avoid the default number to string coercion, since this is a signed conversion. The coercion is used for string concatenation and all standard library functions which accept string arguments (such as `print()` or `io.write()`).

Conditionals

If you're transcribing some code from C/C++, watch out for bit operations in conditionals. In C/C++ any non-zero value is implicitly considered as `true`. E.g. this C code:

```
if (x & 3) ...
```

must not be turned into this Lua code:

```
if band(x, 3) then ... -- wrong!
```

In Lua all objects except `nil` and `false` are considered `true`. This includes all numbers. An explicit comparison against zero is required in this case:

```
if band(x, 3) ~= 0 then ... -- correct!
```

Comparing Against Hex Literals

Comparing the results of bitwise operations (signed numbers) against hex literals (unsigned numbers) needs some additional care. The following conditional expression may or may not work right, depending on the platform you run it on:

```
bit.bor(x, 1) == 0xffffffff
```

E.g. it's never true on a Lua installation with the default number type. Some simple solutions:

Never use hex literals larger than 0x7fffffff in comparisons:

```
bit.bor(x, 1) == -1
```

Or convert them with bit.tobit() before comparing:

```
bit.bor(x, 1) == bit.tobit(0xffffffff)
```

Or use a generic workaround with bit.bxor():

```
bit.bxor(bit.bor(x, 1), 0xffffffff) == 0
```

Or use a case-specific workaround:

```
bit.rshift(x, 1) == 0x7fffffff
```

OPERATIONAL SEMANTICS AND RATIONALE

lua-bit-semantics

Input and Output Ranges

lua-bit-io-ranges

Bitwise operations cannot sensibly be applied to FP numbers (or their underlying bit patterns). They must be converted to integers before operating on them and then back to FP numbers.

It's desirable to define semantics that work the same across all platforms. This dictates that all operations are based on the common denominator of 32 bit integers. The `float` type provides only 24 bits of precision. This makes it unsuitable for use in bitwise operations. Lua BitOp refuses to compile against a Lua installation with this number type.

Bit operations only deal with the underlying bit patterns and generally ignore signedness (except for arithmetic right-shift). They are commonly displayed and treated like unsigned numbers, though.

But the Lua number type must be signed and may be limited to 32 bits. Defining the result type as an unsigned number would not be cross-platform safe. All bit operations are thus defined to return results in the range of signed 32 bit numbers (converted to the Lua number type).

lua-bit-hex-literals

Hexadecimal literals are treated as unsigned numbers by the Lua parser before converting them to the Lua number type. This means they can be out of the range of signed 32 bit integers if the Lua number type has a greater range. E.g. 0xffffffff has a value of 4294967295 in the default installation, but may be -1 on embedded systems. It's highly desirable that hex literals are treated uniformly across systems when used in bitwise operations. All bit operations accept arguments in the signed or the unsigned 32 bit range (and more, see below). Numbers with the same underlying bit pattern are treated the same by all operations.

Modular Arithmetic

lua-bit-modular-arith

Arithmetic operations on n-bit integers are usually based on the rules of modular arithmetic modulo 2^n. Numbers wrap around when the mathematical result of operations is outside their defined range. This simplifies hardware implementations and some algorithms actually require this behavior (like many cryptographic functions).

E.g. for 32 bit integers the following holds: `0xffffffff + 1 = 0`

Arithmetic modulo 2^32 is trivially available if the Lua number type is a 32 bit integer. Otherwise normalization steps must be inserted. Modular arithmetic should work the same across all platforms as far as possible:

- For the default number type of double, arguments can be in the range of ±2^51 and still be safely normalized across all platforms by taking their least-significant 32 bits. The limit is derived from the way doubles are converted to integers.
- The function bit.tobit can be used to explicitly normalize numbers to implement modular addition or subtraction. E.g.

```
              bit.tobit(0xffffffff + 1)
returns 0 on all platforms.
```

- The limit on the argument range implies that modular multiplication is usually restricted to multiplying already normalized numbers with small constants. FP numbers are limited to 53 bits of precision, anyway. E.g. (2^30+1)^2 does not return an odd number when computed with doubles.

BTW: The `tr_i` function shown here lua-bit-shortcuts is one of the non-linear functions of the (flawed) MD5 cryptographic hash and relies on modular arithmetic for correct operation. The result is fed back to other bitwise operations (not shown) and does not need to be normalized until the last step.

Restricted and undefined behaviors

lua-bit-restrictions

The following rules are intended to give a precise and useful definition (for the programmer), yet give the implementation (interpreter and compiler) the maximum flexibility and the freedom to apply advanced optimizations. It's strongly advised not to rely on undefined or implementation-defined behavior.

- All kinds of floating-point numbers are acceptable to the bitwise operations. None of them cause an error, but some may invoke undefined behavior:
  - `-0` is treated the same as `+0` on input and is never returned as a result.
  - Passing `±Inf`, `NaN` or numbers outside the range of `±2^51` as input yields an undefined result.
  - Non-integral numbers may be rounded or truncated in an implementation-defined way. This means the result could differ between different BitOp versions, different Lua VMs, on different platforms or even between interpreted vs. compiled code (as in LuaJIT). Avoid passing fractional numbers to bitwise functions. Use `math.floor()` or `math.ceil()` to get defined behavior.
- Lua provides auto-coercion of string arguments to numbers by default. This behavior is deprecated for bitwise operations.

COPYRIGHT

Lua BitOp is Copyright (C) 2008-2025 Mike Pall. Lua BitOp is free software, released under the MIT license.

Guide to using Lua in Nvim

Introduction

This guide introduces the basics of everyday Lua usage for configuring and controlling Nvim. It assumes some familiarity with the (non-Lua) basics of Nvim (commands, options, mappings, autocommands), which are covered in the user-manual.

This is not a comprehensive encyclopedia of all available features. Think of it as a survival kit: the bare minimum needed to comfortably get started on using Lua in Nvim.

- See lua-plugin for guidance on developing Lua plugins.
- See luaref and lua-concepts for details on the Lua programming language.

About the API

lua-guide-api

The purpose of this guide is to introduce the different ways of interacting with Nvim through Lua (the "API"). This API consists of three different layers:

1. The "Vim API" inherited from Vim: Ex-commands and vimscript-functions as well as user-functions in Vimscript. These are accessed through vim.cmd() and vim.fn respectively, which are discussed under lua-guide-vimscript below.
2. The "Nvim API" written in C for use in remote plugins and GUIs; see api. These functions are accessed through vim.api.
3. The "Lua API" written in and specifically for Lua. These are any other functions accessible through `vim.*` not mentioned already; see lua-stdlib.

This distinction is important, as API functions inherit behavior from their original layer: For example, Nvim API functions always need all arguments to be specified even if Lua itself allows omitting arguments (which are then passed as `nil`); and Vim API functions can use 0-based indexing even if Lua arrays are 1-indexed by default.

Through this, any possible interaction can be done through Lua without writing a complete new API from scratch. For this reason, functions are usually not duplicated between layers unless there is a significant benefit in functionality or performance (e.g., you can map Lua functions directly through nvim_create_autocmd() but not through :autocmd). In case there are multiple ways of achieving the same thing, this guide will only cover what is most convenient to use from Lua.

Using Lua

lua-guide-using-Lua

To run Lua code from the Nvim command line, use the :lua command:

```
:lua print("Hello!")
```

Note: each :lua command has its own scope and variables declared with the local keyword are not accessible outside of the command. This won't work:

```
:lua local foo = 1
:lua print(foo)
" prints "nil" instead of "1"
```

You can also use `:lua=`, which is equivalent to `:lua vim.print(...)`, to conveniently check the value of a variable or a table:

```
:lua =package
```

To run a Lua script in an external file, you can use the :source command exactly like for a Vimscript file:

```
:source ~/programs/baz/myluafile.lua
```

Finally, you can include Lua code in a Vimscript file by putting it inside a :lua-heredoc block:

```
lua << EOF
  local tbl = {1, 2, 3}
  for k, v in ipairs(tbl) do
    print(v)
  end
EOF
```

lua-guide-interrupt

Lua cannot normally be interrupted by CTRL-C (Lua does not implicitly "yield" to Nvim). To allow the user to interrupt potentially slow loops, periodically call vim.wait(). Passing `interval=0` skips the dummy timer setup, minimizing per-call overhead in a tight loop:

```
while true do
  -- ...work...
  local _, code = vim.wait(0, nil, 0)
  if code == -2 then -- CTRL-C.
    break
  end
end
```

See also: https://github.com/neovim/neovim/issues/6800

Using Lua files on startup

lua-guide-config

Nvim supports using `init.vim` or `init.lua` as the configuration file, but not both at the same time. This should be placed in your config directory (run `:echo stdpath('config')` to see where it is). Note that you can also use Lua in `init.vim` and Vimscript in `init.lua`, which will be covered below.

If you'd like to run any other Lua script on startup automatically, then you can simply put it in `plugin/` in your 'runtimepath'.

Lua modules

lua-guide-modules

If you want to load Lua files on demand, you can place them in the `lua/` directory in your 'runtimepath' and load them with `require`. (This is the Lua equivalent of Vimscript's autoload mechanism.)

Let's assume you have the following directory structure:

```
~/.config/nvim
|-- after/
|-- ftplugin/
|-- lua/
|   |-- myluamodule.lua
|   |-- other_modules/
|       |-- anothermodule.lua
|       |-- init.lua
|-- plugin/
|-- syntax/
|-- init.vim
```

Then the following Lua code will load `myluamodule.lua`:

```
require("myluamodule")
```

Note the absence of a `.lua` extension.

Similarly, loading `other_modules/anothermodule.lua` is done via

```
require('other_modules/anothermodule')
-- or
require('other_modules.anothermodule')
```

Note how "submodules" are just subdirectories; the `.` is equivalent to the path separator `/` (even on Windows).

A folder containing an init.lua file can be required directly, without having to specify the name of the file:

```
require('other_modules') -- loads other_modules/init.lua
```

Requiring a nonexistent module or a module which contains syntax errors aborts the currently executing script. `pcall()` may be used to catch such errors. The following example tries to load the `module_with_error` and only calls one of its functions if this succeeds and prints an error message otherwise:

```
local ok, mymod = pcall(require, 'module_with_error')
if not ok then
  print("Module had an error")
else
  mymod.func()
end
```

In contrast to :source, require() not only searches through all `lua/` directories under 'runtimepath', it also caches the module on first use. Calling `require()` a second time will therefore _not_ execute the script again and instead return the cached file. To rerun the file, you need to remove it from the cache manually first:

```
package.loaded['myluamodule'] = nil
require('myluamodule')    -- read and execute the module again from disk
```

See also:
- lua-module-load: how `require()` finds modules
- pcall()

Using Vim commands and functions from Lua

lua-guide-vimscript

All Vim commands and functions are accessible from Lua.

Vim commands

lua-guide-vim-commands

To run an arbitrary Vim command from Lua, pass it as a string to vim.cmd():

```
vim.cmd("colorscheme habamax")
```

Note that special characters will need to be escaped with backslashes:

```
vim.cmd("%s/\\Vfoo/bar/g")
```

An alternative is to use a literal string (see lua-literal) delimited by double brackets `[[ ]]` as in

```
vim.cmd([[%s/\Vfoo/bar/g]])
```

Another benefit of using literal strings is that they can be multiple lines; this allows you to pass multiple commands to a single call of vim.cmd():

```
vim.cmd([[
  highlight Error guibg=red
  highlight link Warning Error
]])
```

This is the converse of :lua-heredoc and allows you to include Vimscript code in your `init.lua`.

If you want to build your Vim command programmatically, the following form can be useful (all these are equivalent to the corresponding line above):

```
vim.cmd.colorscheme("habamax")
vim.cmd.highlight({ "Error", "guibg=red" })
vim.cmd.highlight({ "link", "Warning", "Error" })
```

Vimscript functions

lua-guide-vim-functions

Use vim.fn to call Vimscript functions from Lua. Data types between Lua and Vimscript are automatically converted:

```
print(vim.fn.printf('Hello from %s', 'Lua'))

local reversed_list = vim.fn.reverse({ 'a', 'b', 'c' })
vim.print(reversed_list) -- { "c", "b", "a" }

local function print_stdout(chan_id, data, name)
  print(data[1])
end

vim.fn.jobstart('ls', { on_stdout = print_stdout })
```

This works for both vimscript-functions and user-functions.

Note that hashes (#) are not valid characters for identifiers in Lua, so, e.g., autoload functions have to be called with this syntax:

```
vim.fn['my#autoload#function']()
```

See also:
- vimscript-functions: descriptions of all Vimscript functions
- function-list: Vimscript functions grouped by topic
- :runtime: run all Lua scripts matching a pattern in 'runtimepath'

Variables

lua-guide-variables

Variables can be set and read using the following wrappers, which directly correspond to their variable-scope:

- vim.g: global variables (g:)
- vim.b: variables for the current buffer (b:)
- vim.w: variables for the current window (w:)
- vim.t: variables for the current tabpage (t:)
- vim.v: predefined Vim variables (v:)
- vim.env: environment variables defined in the editor session

Data types are converted automatically. For example:

```
vim.g.some_global_variable = {
  key1 = "value",
  key2 = 300
}

vim.print(vim.g.some_global_variable)
--> { key1 = "value", key2 = 300 }
```

You can target specific buffers (via number), windows (via window-ID), or tabpages by indexing the wrappers:

```
vim.b[2].myvar = 1               -- set myvar for buffer number 2
vim.w[1005].myothervar = true    -- set myothervar for window ID 1005
```

Some variable names may contain characters that cannot be used for identifiers in Lua. You can still manipulate these variables by using the syntax

```
vim.g['my#variable'] = 1
```

Note that you cannot directly change fields of array variables. This won't work:

```
vim.g.some_global_variable.key2 = 400
vim.print(vim.g.some_global_variable)
--> { key1 = "value", key2 = 300 }
```

Instead, you need to create an intermediate Lua table and change this:

```
local temp_table = vim.g.some_global_variable
temp_table.key2 = 400
vim.g.some_global_variable = temp_table
vim.print(vim.g.some_global_variable)
--> { key1 = "value", key2 = 400 }
```

To delete a variable, simply set it to `nil`:

```
vim.g.myvar = nil
```

See also:
- lua-vim-variables

Options lua-guide-options

There are two complementary ways of setting options via Lua.

vim.opt

The most convenient way for setting global and local options, e.g., in `init.lua`, is through `vim.opt` and friends:

- vim.opt: behaves like :set
- vim.opt_global: behaves like :setglobal
- vim.opt_local: behaves like :setlocal

For example, the Vimscript commands

```
set smarttab
set nosmarttab
```

are equivalent to

```
vim.opt.smarttab = true
vim.opt.smarttab = false
```

In particular, they allow an easy way to working with list-like, map-like, and set-like options through Lua tables: Instead of

```
set wildignore=*.o,*.a,__pycache__
set listchars=space:_,tab:>~
set formatoptions=njt
```

you can use

```
vim.opt.wildignore = { '*.o', '*.a', '__pycache__' }
vim.opt.listchars = { space = '_', tab = '>~' }
vim.opt.formatoptions = { n = true, j = true, t = true }
```

These wrappers also come with methods that work similarly to their :set+=, :set^= and :set-= counterparts in Vimscript:

```
vim.opt.shortmess:append({ I = true })
vim.opt.wildignore:prepend('*.o')
vim.opt.whichwrap:remove({ 'b', 's' })
```

The price to pay is that you cannot access the option values directly but must use vim.opt:get():

```
print(vim.opt.smarttab)
--> {...} (big table)
print(vim.opt.smarttab:get())
--> false
vim.print(vim.opt.listchars:get())
--> { space = '_', tab = '>~' }
```

vim.o

For this reason, there exists a more direct variable-like access using `vim.o` and friends, similarly to how you can get and set options via `:echo &number` and `:let &listchars='space:_,tab:>~'`:

- vim.o: behaves like :set
- vim.go: behaves like :setglobal
- vim.bo: for buffer-scoped options
- vim.wo: for window-scoped options (can be double indexed)

For example:

```
vim.o.smarttab = false -- :set nosmarttab
print(vim.o.smarttab)
--> false
vim.o.listchars = 'space:_,tab:>~' -- :set listchars='space:_,tab:>~'
print(vim.o.listchars)
--> 'space:_,tab:>~'
vim.o.isfname = vim.o.isfname .. ',@-@' -- :set isfname+=@-@
print(vim.o.isfname)
--> '@,48-57,/,.,-,_,+,,,#,$,%,~,=,@-@'
vim.bo.shiftwidth = 4 -- :setlocal shiftwidth=4
print(vim.bo.shiftwidth)
--> 4
```

Just like variables, you can specify a buffer number or window-ID for buffer and window options, respectively. If no number is given, the current buffer or window is used:

```
vim.bo[4].expandtab = true -- sets expandtab to true in buffer 4
vim.wo.number = true       -- sets number to true in current window
vim.wo[0].number = true    -- same as above
vim.wo[0][0].number = true -- sets number to true in current buffer
                           -- in current window only
print(vim.wo[0].number)    --> true
```

See also:
- lua-options

Mappings lua-guide-mappings

You can map either Vim commands or Lua functions to key sequences.

Creating mappings lua-guide-mappings-set

Mappings can be created using vim.keymap.set(). This function takes three mandatory arguments:
- {mode} is a string or a table of strings containing the mode prefix for which the mapping will take effect. The prefixes are the ones listed in :map-modes, or "!" for :map!, or empty string for :map.
- {lhs} is a string with the key sequences that should trigger the mapping.
- {rhs} is either a string with a Vim command or a Lua function that should be executed when the {lhs} is entered. An empty string is equivalent to <Nop>, which disables a key.

Examples:

```
-- Normal mode mapping for Vim command
vim.keymap.set('n', '<Leader>ex1', '<cmd>echo "Example 1"<cr>')
-- Normal and Command-line mode mapping for Vim command
vim.keymap.set({'n', 'c'}, '<Leader>ex2', '<cmd>echo "Example 2"<cr>')
-- Normal mode mapping for Lua function
vim.keymap.set('n', '<Leader>ex3', vim.treesitter.start)
-- Normal mode mapping for Lua function with arguments
vim.keymap.set('n', '<Leader>ex4', function() print('Example 4') end)
```

You can map functions from Lua modules via

```
vim.keymap.set('n', '<Leader>pl1', require('plugin').action)
```

Note that this loads the plugin at the time the mapping is defined. If you want to defer the loading to the time when the mapping is executed (as for autoload functions), wrap it in `function() end`:

```
vim.keymap.set('n', '<Leader>pl2', function() require('plugin').action() end)
```

The fourth, optional, argument is a table with keys that modify the behavior of the mapping such as those from :map-arguments. The following are the most useful options:
- `buf`: If given, only set the mapping for the buffer with the specified number; `0` or `true` means the current buffer.

  ```
  -- set mapping for the current buffer
  vim.keymap.set('n', '<Leader>pl1', require('plugin').action, { buf = 0 })
  -- set mapping for the buffer number 4
  vim.keymap.set('n', '<Leader>pl1', require('plugin').action, { buf = 4 })
  ```

- `silent`: If set to `true`, suppress output such as error messages.

  ```
  vim.keymap.set('n', '<Leader>pl1', require('plugin').action, { silent = true })
  ```

- `expr`: If set to `true`, do not execute the {rhs} but use the return value as input. Special keycodes are converted automatically. For example, the following mapping replaces <down> with <c-n> in the popupmenu only:

  ```
  vim.keymap.set('c', '<down>', function()
    if vim.fn.pumvisible() == 1 then return '<c-n>' end
    return '<down>'
  end, { expr = true })
  ```

- `desc`: A string that is shown when listing mappings with, e.g., :map. This is useful since Lua functions as {rhs} are otherwise only listed as `Lua: <number> <source file>:<line>`. Plugins should therefore always use this for mappings they create.

  ```
  vim.keymap.set('n', '<Leader>pl1', require('plugin').action,
    { desc = 'Execute action from plugin' })
  ```

- `remap`: By default, all mappings are nonrecursive (i.e., vim.keymap.set() behaves like :noremap). If the {rhs} is itself a mapping that should be executed, set `remap = true`:

  ```
  vim.keymap.set('n', '<Leader>ex1', '<cmd>echo "Example 1"<cr>')
  -- add a shorter mapping
  vim.keymap.set('n', 'e', '<Leader>ex1', { remap = true })
  ```

  Note: <Plug> mappings are always expanded even with the default `remap = false`:

  ```
  vim.keymap.set('n', '[%', '<Plug>(MatchitNormalMultiBackward)')
  ```

Removing mappings lua-guide-mappings-del

A specific mapping can be removed with vim.keymap.del():

```
vim.keymap.del('n', '<Leader>ex1')
vim.keymap.del({'n', 'c'}, '<Leader>ex2', { buf = 0 })
```

See also:
- `vim.api.`nvim_get_keymap(): return all global mapping
- `vim.api.`nvim_buf_get_keymap(): return all mappings for buffer

Autocommands lua-guide-autocommands

An autocommand is a Vim command or a Lua function that is automatically executed whenever one or more events are triggered, e.g., when a file is read or written, or when a window is created. These are accessible from Lua through the Nvim API.

Creating autocommands lua-guide-autocommand-create

Autocommands are created using `vim.api.`nvim_create_autocmd(), which takes two mandatory arguments:
- `{event}`: a string or table of strings containing the event(s) which should trigger the command or function.
- `{opts}`: a table with keys that control what should happen when the event(s) are triggered.

The most important options are:

- `pattern`: A string or table of strings containing the autocmd-pattern. **Note:** Environment variable like `$HOME` and ~ are not automatically expanded; you need to explicitly use `vim.fn.`expand() for this.
- `command`: A string containing a Vim command.
- `callback`: A Lua function.

You must specify one and only one of `command` and `callback`. If `pattern` is omitted, it defaults to `pattern = '*'`. Examples:

```
vim.api.nvim_create_autocmd({"BufEnter", "BufWinEnter"}, {
  pattern = {"*.c", "*.h"},
  command = "echo 'Entering a C or C++ file'",
})

-- Same autocommand written with a Lua function instead
vim.api.nvim_create_autocmd({"BufEnter", "BufWinEnter"}, {
  pattern = {"*.c", "*.h"},
  callback = function() print("Entering a C or C++ file") end,
})

-- User event triggered by MyPlugin
vim.api.nvim_create_autocmd("User", {
  pattern = "MyPlugin",
  callback = function() print("My Plugin Works!") end,
})
```

Nvim will always call a Lua function with a single table containing information about the triggered autocommand. The most useful keys are
- `match`: a string that matched the `pattern` (see <amatch>)
- `buf`: the number of the buffer the event was triggered in (see <abuf>)
- `file`: the file name of the buffer the event was triggered in (see <afile>)
- `data`: a table with other relevant data that is passed for some events

For example, this allows you to set buffer-local mappings for some filetypes:

```
vim.api.nvim_create_autocmd("FileType", {
  pattern = "lua",
  callback = function(ev)
    vim.keymap.set('n', 'K', vim.lsp.buf.hover, { buf = ev.buf })
  end
})
```

This means that if your callback itself takes an (even optional) argument, you must wrap it in `function() end` to avoid an error:

```
vim.api.nvim_create_autocmd('TextYankPost', {
  callback = function() vim.hl.hl_op() end
})
```

(Since unused arguments can be omitted in Lua function definitions, this is equivalent to `function(args) ... end`.)

Instead of using a pattern, you can create a buffer-local autocommand (see autocmd-buflocal) with `buffer`; in this case, `pattern` cannot be used:

```
-- set autocommand for current buffer
vim.api.nvim_create_autocmd("CursorHold", {
  buffer = 0,
  callback = function() print("hold") end,
})

-- set autocommand for buffer number 33
vim.api.nvim_create_autocmd("CursorHold", {
  buffer = 33,
  callback = function() print("hold") end,
})
```

Similarly to mappings, you can (and should) add a description using `desc`:

```
vim.api.nvim_create_autocmd('TextYankPost', {
  callback = function() vim.hl.hl_op() end,
  desc = "Briefly highlight yanked text"
})
```

Finally, you can group autocommands using the `group` key; this will be covered in detail in the next section.

Grouping autocommands

lua-guide-autocommands-group

Autocommand groups can be used to group related autocommands together; see autocmd-groups. This is useful for organizing autocommands and especially for preventing autocommands to be set multiple times.

Groups can be created with `vim.api.`nvim_create_augroup(). This function takes two mandatory arguments: a string with the name of a group and a table determining whether the group should be cleared (i.e., all grouped autocommands removed) if it already exists. The function returns a number that is the internal identifier of the group. Groups can be specified either by this identifier or by the name (but only if the group has been created first).

For example, a common Vimscript pattern for autocommands defined in files that may be reloaded is

```
augroup vimrc
  " Remove all vimrc autocommands
  autocmd!
  au BufNewFile,BufRead *.html set shiftwidth=4
  au BufNewFile,BufRead *.html set expandtab
augroup END
```

This is equivalent to the following Lua code:

```
local mygroup = vim.api.nvim_create_augroup('vimrc', { clear = true })
vim.api.nvim_create_autocmd({ 'BufNewFile', 'BufRead' }, {
  pattern = '*.html',
  group = mygroup,
  command = 'set shiftwidth=4',
})
vim.api.nvim_create_autocmd({ 'BufNewFile', 'BufRead' }, {
  pattern = '*.html',
  group = 'vimrc',  -- equivalent to group=mygroup
  command = 'set expandtab',
})
```

Autocommand groups are unique for a given name, so you can reuse them, e.g., in a different file:

```
local mygroup = vim.api.nvim_create_augroup('vimrc', { clear = false })
vim.api.nvim_create_autocmd({ 'BufNewFile', 'BufRead' }, {
  pattern = '*.c',
  group = mygroup,
  command = 'set noexpandtab',
})
```

Deleting autocommands

lua-guide-autocommands-delete

You can use `vim.api.`nvim_clear_autocmds() to remove autocommands. This function takes a single mandatory argument that is a table of keys describing the autocommands that are to be removed:

```
-- Delete all BufEnter and InsertLeave autocommands
vim.api.nvim_clear_autocmds({event = {"BufEnter", "InsertLeave"}})

-- Delete all autocommands that uses "*.py" pattern
vim.api.nvim_clear_autocmds({pattern = "*.py"})

-- Delete all autocommands in group "scala"
vim.api.nvim_clear_autocmds({group = "scala"})

-- Delete all ColorScheme autocommands in current buffer
vim.api.nvim_clear_autocmds({event = "ColorScheme", buffer = 0 })
```

Note: Autocommands in groups will only be removed if the `group` key is specified, even if another option matches it.

See also
- nvim_get_autocmds(): return all matching autocommands
- nvim_exec_autocmds(): execute all matching autocommands

User commands

lua-guide-commands

user-commands are custom Vim commands that call a Vimscript or Lua function. Just like built-in commands, they can have arguments, act on ranges, or have custom completion of arguments. As these are most useful for plugins, we will cover only the basics of this advanced topic.

Creating user commands

lua-guide-commands-create

User commands can be created via nvim_create_user_command(). This function takes three mandatory arguments:
- a string that is the name of the command (which must start with an uppercase letter to distinguish it from builtin commands);
- a string containing Vim commands or a Lua function that is executed when the command is invoked;
- a table with command-attributes; in addition, it can contain the keys `desc` (a string describing the command); `force` (set to `false` to avoid replacing an already existing command with the same name), and `preview` (a Lua function that is used for :command-preview).

Example:

```
vim.api.nvim_create_user_command('Test', 'echo "It works!"')
vim.cmd.Test()
--> It works!
```

(Note that the third argument is mandatory even if no attributes are given.)

Lua functions are called with a single table argument containing arguments and modifiers. The most important are:

- `name`: a string with the command name
- `fargs`: a table containing the command arguments split by whitespace (see <f-args>)
- `bang`: `true` if the command was executed with a `!` modifier (see <bang>)
- `line1`: the starting line number of the command range (see <line1>)
- `line2`: the final line number of the command range (see <line2>)
- `range`: the number of items in the command range: 0, 1, or 2 (see <range>)
- `count`: any count supplied (see <count>)
- `smods`: a table containing the command modifiers (see <mods>)

For example:

```
vim.api.nvim_create_user_command('Upper',
  function(opts)
    print(string.upper(opts.fargs[1]))
  end,
  { nargs = 1 })

vim.cmd.Upper('foo')
--> FOO
```

The `complete` attribute can take a Lua function in addition to the attributes listed in :command-complete.

```
vim.api.nvim_create_user_command('Upper',
  function(opts)
    print(string.upper(opts.fargs[1]))
  end,
  { nargs = 1,
    complete = function(ArgLead, CmdLine, CursorPos)
      -- return completion candidates as a list-like table
      return { "foo", "bar", "baz" }
    end,
})
```

Buffer-local user commands are created with `vim.api.`nvim_buf_create_user_command(). Here the first argument is the buffer number (`0` being the current buffer); the remaining arguments are the same as for nvim_create_user_command():

```
vim.api.nvim_buf_create_user_command(0, 'Upper',
  function(opts)
    print(string.upper(opts.fargs[1]))
  end,
  { nargs = 1 })
```

Deleting user commands

lua-guide-commands-delete

User commands can be deleted with `vim.api.`nvim_del_user_command(). The only argument is the name of the command:

```
vim.api.nvim_del_user_command('Upper')
```

To delete buffer-local user commands use `vim.api.`nvim_buf_del_user_command(). Here the first argument is the buffer number (`0` being the current buffer), and second is command name:

```
vim.api.nvim_buf_del_user_command(4, 'Upper')
```

Credits

lua-guide-credits

This guide is in large part taken from nanotee's Lua guide: https://github.com/nanotee/nvim-lua-guide

Thank you @nanotee!

Guide to developing Lua plugins for Nvim

Introduction

This document provides guidance for developing Nvim Lua plugins.

- See lua-guide for guidance on using Lua to configure and operate Nvim.
- See luaref and lua-concepts for details on the Lua programming language.

Creating your first plugin lua-plugin-new

Any Vimscript or Lua code file that lives in the right directory, automatically is a "plugin". There's no manifest or "registration" step.

You can try it right now:

1. Visit your config directory:

```
:exe 'edit' stdpath('config')
```

2. Create a `plugin/foo.lua` file in there.
3. Add something to it, like:

```
vim.print('Hello World')
```

4. Start `nvim` and notice that it prints "Hello World" in the messages area. Check `:messages` if you don't see it.

Besides `plugin/foo.lua`, which is always run at startup, you can define Lua modules in the `lua/` directory. Those modules aren't loaded until your `plugin/foo.lua`, or the user, calls `require(…)`.

Type safety lua-plugin-type-safety

Lua, as a dynamically typed language, is great for configuration. It provides virtually immediate feedback. But for larger projects, this can be a double-edged sword, leaving your plugin susceptible to unexpected bugs at the wrong time.

You can leverage LuaCATS or "emmylua" annotations https://luals.github.io/wiki/annotations/ along with lua-language-server ("LuaLS") https://luals.github.io/ to catch potential bugs in your CI before your plugin's users do. The Nvim codebase uses these annotations extensively.

TOOLS

- lua-typecheck-action https://github.com/marketplace/actions/lua-typecheck-action
- lua-language-server https://luals.github.io

Keymaps lua-plugin-keymaps

Avoid creating excessive keymaps automatically. Doing so can conflict with user mappings.

NOTE: An example for uncontroversial keymaps are buffer-local mappings for specific file types or floating windows, or <Plug> mappings.

A common approach to allow keymap configuration is to define a declarative DSL https://en.wikipedia.org/wiki/Domain-specific_language via a `setup` function.

However, doing so means that

- You will have to implement and document it yourself.
- Users will likely face inconsistencies if another plugin has a slightly different DSL.
- init.lua scripts that call such a `setup` function may throw an error if the plugin is not installed or disabled.

As an alternative, you can provide <Plug> mappings to allow users to define their own keymaps with vim.keymap.set().

- This requires one line of code in user configs.
- Even if your plugin is not installed or disabled, creating the keymap won't throw an error.

Another option is to simply expose a Lua function or user-commands.

Some benefits of <Plug> mappings are that you can

- Enforce options like `expr = true`.
- Use vim.keymap's built-in mode handling to expose functionality only for specific map-modes.
- Handle different map-modes differently with a single mapping, without adding mode checks to the underlying implementation.
- Detect user-defined mappings through hasmapto() before creating defaults.

Some benefits of exposing a Lua function are:

- Extensibility, if the function takes an options table as an argument.
- A cleaner UX, if there are many options and enumerating all combinations of options would result in a lot of <Plug> mappings.

NOTE: If your function takes an options table, users may still benefit from <Plug> mappings for the most common combinations.

KEYMAP EXAMPLE

In your plugin:

```
vim.keymap.set('n', '<Plug>(SayHello)', function()
    print('Hello from normal mode')
end)

vim.keymap.set('v', '<Plug>(SayHello)', function()
    print('Hello from visual mode')
end)
```

In the user's config:

```
vim.keymap.set({'n', 'v'}, '<leader>h', '<Plug>(SayHello)')
```

Initialization lua-plugin-init

Strictly separated configuration and smart initialization allow your plugin to work out of the box. Common approaches are:

- A Lua function, e.g. `setup(opts)` or `configure(opts)`, which only overrides the default configuration and does not contain any initialization logic.
- A Vimscript compatible table (e.g. in the vim.g or vim.b namespace) that your plugin reads from and validates at initialization time. See also lua-vim-variables.

Typically, automatic initialization logic is done in a plugin or ftplugin script. See also 'runtimepath'.

On the other hand, a single `setup(opts)` that combines configuration and initialization may be useful in specific cases:

- Customizing complex initialization, where there is a significant risk of misconfiguration.
- Requiring users to opt in for plugin functionality that should not be initialized automatically.

Keep in mind that this approach requires users to call `setup` in order to use your plugin, even if the default configuration is enough for them. Consider carefully whether your plugin benefits from combined `setup()` pattern before adopting it. This article chronicles the history and tradeoffs of `setup()`: https://mrcjkb.dev/posts/2023-08-22-setup.html

NOTE: A well designed plugin has minimal impact on startup time. See also lua-plugin-lazy.

Lazy loading lua-plugin-lazy

Some users like to micro-manage "lazy loading" of plugins by explicitly configuring which commands and key mappings load the plugin.

Your plugin should not depend on every user micro-managing their configuration in such a way. Nvim has a mechanism for every plugin to do its own implicit lazy-loading, via autoload (Vimscript) and `lua/` (Lua).

Plugin authors can provide "lazy loading" by providing a `plugin/<name>.lua` file which defines their commands and keymappings. This file should be small, and should not eagerly `require()` the rest of your plugin. Commands and mappings should do the `require()`.

Guidance:

- Plugins should arrange their "lazy" behavior once, instead of expecting every user to micromanage it.
- Keep `plugin/<name>.lua` small, avoid eagerly calling `require()` on modules until a command or mapping is actually used.

Defer require() calls lua-plugin-defer-require

`plugin/<name>.lua` scripts (plugin) are eagerly run at startup; this is intentional, so that plugins can setup the (minimal) commands and keymappings that users will use to invoke the plugin. This also means these "plugin/" files should NOT eagerly `require` Lua modules.

For example, instead of:

```
local foo = require('foo')
vim.api.nvim_create_user_command('MyCommand', function()
    foo.do_something()
end, {
  -- ...
})
```

which calls `require('foo')` as soon as the module is loaded, you can lazy-load it by moving the `require` into the command's implementation:

```
vim.api.nvim_create_user_command('MyCommand', function()
    local foo = require('foo')
    foo.do_something()
end, {
  -- ...
})
```

Likewise, if a plugin uses a Lua module as an entrypoint, it should defer `require` calls too.

NOTE: For a Vimscript alternative to `require`, see autoload.

NOTE: If you are worried about eagerly creating user commands, autocommands or keymaps at startup: Plugin managers that provide abstractions for lazy-loading plugins on such events do the same amount of work. There is no performance benefit for users to define lazy-loading entrypoints in their configuration instead of plugins defining it in `plugin/<name>.lua`.

NOTE: You can use --startuptime to profile the impact a plugin has on startup time.

Filetype-specific functionality lua-plugin-filetype

Consider making use of 'filetype' for any functionality that is specific to a filetype, by putting the initialization logic in a `ftplugin/{filetype}.lua` script.

For buffers owned by your plugin (often used to show a custom UI or view), typically your plugin will set a custom 'filetype'. In that case, it's useful to set the 'filetype' "as late as possible", so that users can override buffer-local settings after your plugin has (re)initialized the buffer.

FILETYPE EXAMPLE

A plugin tailored to Rust development might have initialization in `ftplugin/rust.lua`:

```
if not vim.g.loaded_my_rust_plugin then
    -- Initialize
end
-- NOTE: Using `vim.g.loaded_` prevents the plugin from initializing twice
-- and allows users to prevent plugins from loading
-- (in both Lua and Vimscript).
vim.g.loaded_my_rust_plugin = true

local bufnr = vim.api.nvim_get_current_buf()
-- do something specific to this buffer,
-- e.g. add a |<Plug>| mapping or create a command
vim.keymap.set('n', '<Plug>(MyPluginBufferAction)', function()
    print('Hello')
end, { buf = bufnr })
```

Opening files lua-plugin-open-file

To robustly open a file, see open-file.

Configuration lua-plugin-config

To allow users to override buffer-local configuration for filetypes owned by your plugin, publish a FileType event, "as late as possible". lua-plugin-filetype

Once you have merged the default configuration with the user's config, you should validate configs.

Validations could include:

- Correct types, see vim.validate()
- Unknown fields in the user config (e.g. due to typos). This can be tricky to implement, and may be better suited for a health check, to reduce overhead.

UI lua-plugin-ui

Some plugins have their own "UI" which they present in a buffer that the plugin "owns". In that buffer typically you will want to provide custom actions.

Besides creating <Plug> mappings, you may want to consider providing actions by defining an in-process LSP server. Offering actions as code-actions vim.lsp.buf.code_action() means the user can see all available actions using the default gra mapping to view the code-actions menu. They can even define mappings to a specific action by invoking `vim.lsp.buf.code_action()` with the `filter` + `apply` parameters:

```
vim.lsp.buf.code_action({
    apply = true,
    filter = function(a)
        return a.title == 'Do something'
    end,
})
```

Example: See `runtime/lua/vim/pack/_lsp.lua` for how vim.pack defines an in-process LSP server to provide interactive features in its `nvim-pack://confirm` buffer.

Troubleshooting lua-plugin-troubleshooting

While developing a plugin, you can use the :restart command to see the result of code changes in your plugin.

HEALTH

Nvim's "health" framework gives plugins a simple way to report status checks to users. See health-dev for an example.

Basically, this just means your plugin will have a `lua/{plugin}/health.lua` file. :checkhealth will automatically find this file when it runs.

Some things to validate:

- User configuration
- Proper initialization
- Presence of Lua dependencies (e.g. other plugins)
- Presence of external dependencies

MINIMAL CONFIG TEMPLATE

It can be useful to provide a template for a minimal configuration, along with a guide on how to use it to reproduce issues.

Versioning and releases lua-plugin-versioning

Consider:

- Use vim.deprecate() or a `---@deprecate` annotation when you need to communicate a (future) breaking change or discouraged practice.
- Using SemVer https://semver.org/ tags and releases to properly communicate bug fixes, new features, and breaking changes.
- Automating versioning and releases in CI.
- Publishing to luarocks https://luarocks.org, especially if your plugin has dependencies or components that need to be built; or if it could be a dependency for another plugin.

FURTHER READING

- Luarocks ❤️ Nvim https://github.com/nvim-neorocks/sample-luarocks-plugin

VERSIONING TOOLS

- luarocks-tag-release https://github.com/marketplace/actions/luarocks-tag-release
- release-please-action https://github.com/marketplace/actions/release-please-action
- semantic-release https://github.com/semantic-release/semantic-release

Documentation **lua-plugin-doc**

Provide vimdoc (see help-writing), so that users can read your plugin's documentation in Nvim, by entering `:h {plugin}` in command-mode. The help-tags (the right-aligned "search keywords" in the help documents) are regenerated using the :helptags command.

DOCUMENTATION TOOLS

- panvimdoc https://github.com/kdheepak/panvimdoc

Lua engine Lua

INTRODUCTION lua-intro

The Lua 5.1 script engine is builtin and always available. Try this command to get an idea of what lurks beneath:

```
:lua vim.print(package.loaded)
```

Nvim includes a "standard library" lua-stdlib for Lua. It complements the "editor stdlib" (vimscript-functions + Ex-commands) and the API, all of which can be used from Lua code (lua-vimscript vim.api). These three namespaces form the Nvim programming interface.

Lua plugins and user config are automatically discovered and loaded, just like Vimscript. See lua-guide for practical guidance.

You can also run Lua scripts from your shell using the -l argument:

```
nvim -l foo.lua [args...]
```

lua-compat

Lua 5.1 is the permanent interface for Nvim Lua. Plugins should target Lua 5.1 as specified in luaref; later versions (which are essentially different, incompatible, dialects) are not supported. This includes extensions such as `goto` that some Lua 5.1 interpreters like LuaJIT may support.

lua-luajit

While Nvim officially only requires Lua 5.1 support, it should be built with LuaJIT or a compatible fork on supported platforms for performance reasons. LuaJIT also comes with useful extensions such as `ffi`, lua-profile, and enhanced standard library functions; these cannot be assumed to be available, and Lua code in init.lua or plugins should check the `jit` global variable before using them:

```
if jit then
  -- code for luajit
else
  -- code for plain lua 5.1
end
```

One exception is the LuaJIT `bit` extension, which is always available: when built with PUC Lua, Nvim includes a fallback implementation which provides `require("bit")`. See lua-bit.

lua-profile

If Nvim is built with LuaJIT, Lua code can be profiled via

```
-- Start a profiling session:
require('jit.p').start('ri1', '/tmp/profile')

-- Perform arbitrary tasks (use plugins, scripts, etc.) ...

-- Stop the session. Profile is written to /tmp/profile.
require('jit.p').stop()
```

See https://luajit.org/ext_profiler.html or the `p.lua` source for details:

```
:lua vim.cmd.edit(package.searchpath('jit.p', package.path))
```

LUA CONCEPTS AND IDIOMS lua-concepts

Lua is very simple, and _consistent_: while there are some quirks, once you internalize those quirks, everything works the same everywhere. Scopes (closures) in particular are very consistent, unlike JavaScript or most other languages.

Lua has three fundamental mechanisms—one for "each major aspect of programming": tables, closures, and coroutines.
https://www.lua.org/doc/cacm2018.pdf

- Tables are the "object" or container datastructure: they represent both lists and maps, you can extend them to represent your own datatypes and change their behavior using metatables (like Python's "datamodel").
- EVERY scope in Lua is a closure: a function is a closure, a module is a closure, a do block (lua-do) is a closure--and they all work the same. A Lua module is literally just a big closure discovered on the "path" (where your modules are found: package.cpath).
- Stackful coroutines enable cooperative multithreading, generators, and versatile control for both Lua and its host (Nvim).

lua-error-handling

Lua functions may throw lua-errors for exceptional (unexpected) failures, which you can handle with pcall(). lua-result-or-message
When failure is normal and expected, it's idiomatic to return `nil` which signals to the caller that failure is not "exceptional" and must be handled. This "result-or-message" pattern is expressed as the multi-value return type `any|nil,nil|string`, or in LuaLS notation:

```
---@return any|nil    # result on success, nil on failure.
---@return nil|string # nil on success, error message on failure.
```

Examples of the "result-or-message" pattern:

- vim.ui.open()
- io.open()
- luv-error-handling

When a caller can't proceed on failure, it's idiomatic to `assert()` the "result-or-message" result:

```
local value = assert(fn())
```

Guidance: use the "result-or-message" pattern for...

- Functions where failure is expected, especially when communicating with the external world. E.g. HTTP requests or LSP requests often fail because of server problems, even if the caller did everything right.
- Functions that return a value, e.g. Foo:new().
- When there is a list of known error codes which can be returned as a third value (like luv-error-handling).

iterator

An iterator is just a function that can be called repeatedly to get the "next" value of a collection (or any other iterable). This interface is expected by for-in loops, produced by pairs(), supported by vim.iter, etc. https://www.lua.org/pil/7.1.html

iterable

An "iterable" is anything that vim.iter() can consume: tables, dicts, lists, iterator functions, tables implementing the __call() metamethod, and vim.iter() objects.

list-iterator

Iterators on lua-list tables have a "middle" and "end", whereas iterators in general may be logically infinite. Therefore some vim.iter operations (e.g. IterArray:rev()) make sense only on list-like tables (which are finite by definition).

lua-function-call

Lua functions can be called in multiple ways. Consider the function:

```
local foo = function(a, b)
    print("A: ", a)
    print("B: ", b)
end
```

The first way to call this function is:

```
foo(1, 2)
-- ==== Result ====
-- A: 1
-- B: 2
```

This way of calling a function is familiar from most scripting languages. In Lua, any missing arguments are passed as `nil`, and extra parameters are silently discarded. Example:

```
foo(1)
-- ==== Result ====
-- A: 1
-- B: nil
```

kwargs

When calling a function, you can omit the parentheses if the function takes exactly one string literal (`"foo"`) or table literal (`{1,2,3}`). The latter is often used to mimic "named parameters" ("kwargs" or "keyword args") as in languages like Python and C#. Example:

```
local func_with_opts = function(opts)
    local will_do_foo = opts.foo
    local filename = opts.filename
    -- ...
end

func_with_opts { foo = true, filename = "hello.world" }
```

There's nothing special going on here except that parentheses are implicitly added. But visually, this small bit of sugar gets reasonably close to a "keyword args" interface.

lua-regex

Lua intentionally does not support regular expressions, instead it has limited lua-patterns which avoid the performance pitfalls of extended regex. Lua scripts can also use Vim regex via vim.regex().

Examples:

```
print(string.match("foo123bar123", "%d+"))
-- 123
print(string.match("foo123bar123", "[^%d]+"))
-- foo
print(string.match("foo123bar123", "[abc]+"))
-- ba
print(string.match("foo.bar", "%.bar"))
-- .bar
```

lua-truthy

Only `false` and `nil` evaluate to "false" in Lua. All other values are "true".

IMPORTING LUA MODULES

lua-module-load

Modules are searched for under the directories specified in 'runtimepath' and packages-runtimepath, in the order they appear in the output of this command

```
:echo nvim_list_runtime_paths()
```

Any "." in the module name is treated as a directory separator when searching. For a module `foo.bar`, each directory is searched for `lua/foo/bar.lua`, then `lua/foo/bar/init.lua`. If no files are found, the directories are searched again for a shared library with a name matching `lua/foo/bar.?`, where `?` is a list of suffixes (such as `so` or `dll`) derived from the initial value of package.cpath. If still no files are found, Nvim falls back to Lua's default search mechanism. The first script found is run and `require()` returns the value returned by the script if any, else `true`.

The return value is cached after the first call to `require()` for each module, with subsequent calls returning the cached value without searching for, or executing any script. For further details see require().

For example, if 'runtimepath' is `foo,bar` and package.cpath was `./?.so;./?.dll` at startup, `require('mod')` searches these paths in order and loads the first module found ("first wins"):

```
foo/lua/mod.lua
foo/lua/mod/init.lua
bar/lua/mod.lua
bar/lua/mod/init.lua
foo/lua/mod.so
foo/lua/mod.dll
bar/lua/mod.so
bar/lua/mod.dll
```

Note:

- Although 'runtimepath' is tracked, Nvim does not track current values of package.path or package.cpath. If you happen to delete some paths from there you can set 'runtimepath' to trigger an update:

  ```
  let &runtimepath = &runtimepath
  ```

- Skipping paths from 'runtimepath' which contain semicolons applies both to package.path and package.cpath. Given that there are some badly written plugins using shell, which will not work with paths containing semicolons, it is better to not have them in 'runtimepath' at all.

lua-script-location

To get its own location, Lua scripts/modules can use debug.getinfo():

```
debug.getinfo(1, 'S').source:gsub('^@', '')
```

COMMANDS

lua-commands

These commands execute a Lua chunk from either the command line (:lua, :luado) or a file (:luafile) on the given line [range]. As always in Lua, each chunk has its own scope (closure), so only global variables are shared between command calls. The lua-stdlib modules, user modules, and anything else on package.path are available.

The Lua print() function redirects its output to the Nvim message area, with arguments separated by " " (space) instead of "\t" (tab).

:lua= :lua :lua {chunk} Executes Lua chunk {chunk}. If {chunk} starts with "=" the rest of the chunk is evaluated as an expression and printed. `:lua =expr` and `:=expr` are equivalent to `:lua print(vim.inspect(expr))`. E5107 E5108

Examples:

```
:lua vim.api.nvim_command('echo "Hello, Nvim!"')
```

To see the Lua version:

```
:lua print(_VERSION)
```

To see the LuaJIT version:

```
:lua =jit.version
```

:{range}lua Executes buffer lines in {range} as Lua code. Unlike :source, this always treats the lines as Lua code.

Example: select the following code and type ":lua<Enter>" to execute it:

```
print(string.format(
    'unix time: %s', os.time()))
```

:lua-heredoc

:lua << [trim] [{endmarker}] {script} {endmarker} Executes Lua script {script} from within Vimscript. You can omit [endmarker] after the "<<" and use a dot "." after {script} (similar to :append, :insert). Refer to :let-heredoc for more information.

Example:

```
function! CurrentLineInfo()
lua << EOF
local linenr = vim.api.nvim_win_get_cursor(0)[1]
local curline = vim.api.nvim_buf_get_lines(0, linenr - 1, linenr, false)[1]
print(string.format('Line [%d] has %d bytes', linenr, #curline))
EOF
endfunction
```

Note that the `local` variables will disappear when the block finishes. But not globals.

:luado

:[range]luado {body} Executes Lua chunk "function(line, linenr) {body} end" for each buffer line in [range], where `line` is the current line text (without <EOL>), and `linenr` is the current line number. If the function returns a string that becomes the text of the corresponding buffer line. Default [range] is the whole file: "1,$". E5109 E5110 E5111

Examples:

```
:luado return string.format("%s\t%d", line:reverse(), #line)

:lua require"lpeg"
:lua -- balanced parenthesis grammar:
:lua bp = lpeg.P{ "(" * ((1 - lpeg.S"()") + lpeg.V(1))^0 * ")" }
:luado if bp:match(line) then return "=>\t" .. line end
```

:luafile

:luafile {file} Execute Lua script in {file}. The whole argument is used as the filename (like :edit), spaces do not need to be escaped. Alternatively you can :source Lua files. E5111 E5112 E5113

Examples:

```
:luafile script.lua
:luafile %
```

luaeval()

lua-eval

The (dual) equivalent of "vim.eval" for passing Lua values to Nvim is "luaeval". "luaeval" takes an expression string and an optional argument used for _A inside expression and returns the result of the expression. It is semantically equivalent in Lua to:

```
local chunkheader = "local _A = select(1, ...) return "
function luaeval (expstr, arg)
    local chunk = assert(loadstring(chunkheader .. expstr, "luaeval"))
    return chunk(arg) -- return typval
end
```

Lua nils, numbers, strings, tables and booleans are converted to their respective Vimscript types. If a Lua string contains a NUL byte, it will be converted to a Blob. Conversion of other Lua types is an error.

The magic global "_A" contains the second argument to luaeval().

Example:

```
:echo luaeval('_A[1] + _A[2]', [40, 2])
" 42
:echo luaeval('string.match(_A, "[a-z]+")', 'XYXfoo123')
" foo
```

lua-table-ambiguous

Lua tables are used as both dictionaries and lists, so it is impossible to decide whether empty table is a list or a dict. Also Lua does not have integer numbers. To disambiguate these cases, we define: lua-list

1. Empty table is a list. Use vim.empty_dict() to represent empty dict.
2. Table with N consecutive (no nil values, aka "holes") integer keys 1…N is a list. See also list-iterator. lua-dict
3. Table with string keys, none of which contains NUL byte, is a dict.
4. Table with string keys, at least one of which contains NUL byte, is also considered to be a dictionary, but this time it is converted to a msgpack-special-map. lua-special-tbl E5100 E5101 E5102
5. Table with vim.type_idx key may be a dictionary, a list or floating-point value:
   - {[vim.type_idx]=vim.types.float, [vim.val_idx]=1} is converted to a floating-point 1.0. Note that by default integral Lua numbers are converted to Numbers, non-integral are converted to Floats. This variant allows integral Floats.
   - {[vim.type_idx]=vim.types.dictionary} is converted to an empty dictionary, {[vim.type_idx]=vim.types.dictionary, [42]=1, a=2} is converted to a dictionary {'a': 42}: non-string keys are ignored. Without vim.type_idx key tables with keys not fitting in 1., 2. or 3. are errors.
   - {[vim.type_idx]=vim.types.array} is converted to an empty list. As well as {[vim.type_idx]=vim.types.array, [42]=1}: integral keys that do not form a 1-step sequence from 1 to N are ignored, as well as all non-integral keys.

Examples:

```
:echo luaeval('math.pi')
:function Rand(x,y) " random uniform between x and y
:  return luaeval('(_A.y-_A.x)*math.random()+_A.x', {'x':a:x,'y':a:y})
:  endfunction
:echo Rand(1,10)
```

Note: Second argument to luaeval is converted ("marshalled") from Vimscript to Lua, so changes to Lua containers do not affect values in Vimscript. Return value is also always converted. When converting, msgpack-special-dicts are treated specially.

Vimscript v:lua interface

v:lua-call

From Vimscript the special v:lua prefix can be used to call Lua functions which are global or accessible from global tables. The expression

```
call v:lua.func(arg1, arg2)
```

is equivalent to the Lua chunk

```
return func(...)
```

where the args are converted to Lua values. The expression

```
call v:lua.somemod.func(args)
```

is equivalent to the Lua chunk

```
return somemod.func(...)
```

Lua module functions can be accessed like:

```
call v:lua.require'mypack'.func(arg1, arg2)
call v:lua.require'mypack.submod'.func(arg1, arg2)
```

Note: Only single quote form without parens is allowed. Using `require"mypack"` or `require('mypack')` as a prefix does NOT work.

You can use `v:lua` in "func" options like 'tagfunc', 'omnifunc', etc. For example consider the following Lua omnifunc handler:

```
function mymod.omnifunc(findstart, base)
  if findstart == 1 then
    return 0
  else
    return {'stuff', 'steam', 'strange things'}
  end
end
-- Note: The module ("mymod") must be a Lua global, or use require() as
-- shown above to access it from a package.
vim.bo[buf].omnifunc = 'v:lua.mymod.omnifunc'
```

You can also use `v:lua` to call Lua functions as Vimscript methods:

```
:eval arg1->v:lua.somemod.func(arg2)
```

Note: `v:lua` without a call is not allowed in a Vimscript expression: Funcrefs cannot represent Lua functions. The following are errors:

```
let g:Myvar = v:lua.myfunc        " Error
call SomeFunc(v:lua.mycallback)   " Error
let g:foo = v:lua                 " Error
let g:foo = v:['lua']             " Error
```

Lua standard modules

lua-stdlib

The Nvim Lua "standard library" (stdlib) is the `vim` module, which exposes various functions and sub-modules. It is always loaded, thus `require("vim")` is unnecessary.

You can peek at the module properties:

```
:lua vim.print(vim)
```

Result is something like this:

```
{
  _os_proc_children = <function 1>,
  _os_proc_info = <function 2>,
  ...
  api = {
    nvim__id = <function 5>,
    nvim__id_array = <function 6>,
    ...
  },
  deepcopy = <function 106>,
  gsplit = <function 107>,
  ...
}
```

To find documentation on e.g. the "deepcopy" function:

```
:help vim.deepcopy()
```

Note that underscore-prefixed functions (e.g. "_os_proc_children") are internal/private and must not be used by plugins.

VIM.UV

lua-loop **vim.uv**

`vim.uv` exposes the "luv" Lua bindings for the libUV library that Nvim uses for networking, filesystem, and process management, see luvref.txt. In particular, it allows interacting with the main Nvim luv-event-loop.

E5560 lua-loop-callbacks It is an error to directly invoke `vim.api` functions (except api-fast) in `vim.uv` callbacks. For example, this is an error:

```
local timer = vim.uv.new_timer()
timer:start(1000, 0, function()
  vim.api.nvim_command('echomsg "test"')
end)
```

To avoid the error, schedule the callback:

```
local timer = vim.uv.new_timer()
timer:start(1000, 0, vim.schedule_wrap(function()
  vim.api.nvim_command('echomsg "test"')
end))
```

(For one-shot timers, see vim.defer_fn(), which automatically adds the wrapping.)

Example: repeating timer

1. Save this code to a file.
2. Execute it with ":luafile %".

```
-- Create a timer handle (implementation detail: uv_timer_t).
local timer = vim.uv.new_timer()
local i = 0
-- Waits 1000ms, then repeats every 750ms until timer:close().
timer:start(1000, 750, function()
  print('timer invoked! i='..tostring(i))
  if i > 4 then
    timer:close()  -- Always close handles to avoid leaks.
  end
  i = i + 1
end)
print('sleeping');
```

Example: File-change detection watch-file

1. Save this code to a file.
2. Execute it with ":luafile %".
3. Use ":Watch %" to watch any file.
4. Try editing the file from another text editor.
5. Observe that the file reloads in Nvim (because on_change() calls :checktime).

```
local w = vim.uv.new_fs_event()
local function on_change(err, fname, status)
  -- Do work...
  vim.api.nvim_command('checktime')
  -- Debounce: stop/start.
  w:stop()
  watch_file(fname)
end
function watch_file(fname)
  local fullpath = vim.api.nvim_call_function(
    'fnamemodify', {fname, ':p'})
  w:start(fullpath, {}, vim.schedule_wrap(function(...)
    on_change(...) end))
end
vim.api.nvim_command(
  "command! -nargs=1 Watch call luaeval('watch_file(_A)', expand('<args>'))")
```

inotify-limitations

When on Linux you may need to increase the maximum number of `inotify` watches and queued events as the default limit can be too low. To increase the limit, run:

```
sysctl fs.inotify.max_user_watches=494462
```

This will increase the limit to 494462 watches and queued events. These lines can be added to `/etc/sysctl.conf` to make the changes persistent.

Note that each watch is a structure in the Kernel, thus available memory is also a bottleneck for using inotify. In fact, a watch can take up to 1KB of space. This means a million watches could result in 1GB of extra RAM usage.

Example: TCP echo-server tcp-server

1. Save this code to a file.
2. Execute it with ":luafile %".
3. Note the port number.
4. Connect from any TCP client (e.g. "nc 0.0.0.0 36795"):

```
local function create_server(host, port, on_connect)
  local server = vim.uv.new_tcp()
  server:bind(host, port)
  server:listen(128, function(err)
    assert(not err, err)  -- Check for errors.
    local sock = vim.uv.new_tcp()
    server:accept(sock)  -- Accept client connection.
    on_connect(sock)  -- Start reading messages.
  end)
  return server
end
local server = create_server('0.0.0.0', 0, function(sock)
  sock:read_start(function(err, chunk)
    assert(not err, err)  -- Check for errors.
    if chunk then
      sock:write(chunk)  -- Echo received messages to the channel.
    else  -- EOF (stream closed).
      sock:close()  -- Always close handles to avoid leaks.
    end
  end)
end)
print('TCP echo-server listening on port: '..server:getsockname().port)
```

Multithreading lua-loop-threading

Plugins can perform work in separate (os-level) threads using the threading APIs in luv, for instance `vim.uv.new_thread`. Each thread has its own separate Lua interpreter state, with no access to Lua globals on the main thread. Neither can the editor state (buffers, windows, etc) be directly accessed from threads.

A subset of the `vim.*` stdlib is available in threads, including:

- `vim.uv` with a separate event loop per thread.
- `vim.mpack` and `vim.json` (useful for serializing messages between threads)
- `require` in threads can use Lua packages from the global package.path
- `print()` and `vim.inspect`
- `vim.text.diff`
- Most utility functions in `vim.*` that work with pure Lua values, like `vim.split`, `vim.tbl_*`, `vim.list_*`, etc.
- `vim.is_thread()` returns true from a non-main thread.

VIM vim.builtin

vim.api.{func}({...}) vim.api
Invokes Nvim API function `{func}` with arguments `{...}`. Example: call the "nvim_get_current_line()" API function:

```
print(tostring(vim.api.nvim_get_current_line()))
```

vim.NIL vim.NIL
Special value representing NIL in RPC and v:null in Vimscript conversion, and similar cases. Lua `nil` cannot be used as part of a Lua table representing a Dictionary or Array, because it is treated as missing: `{"foo", nil}` is the same as `{"foo"}`.

vim.type_idx vim.type_idx
Type index for use in lua-special-tbl. Specifying one of the values from vim.types allows typing the empty table (it is unclear whether empty Lua table represents empty list or empty array) and forcing integral numbers to be Float. See lua-special-tbl for more details.

vim.val_idx vim.val_idx
Value index for tables representing Floats. A table representing floating-point value 1.0 looks like this:

```
{
  [vim.type_idx] = vim.types.float,
  [vim.val_idx] = 1.0,
}
```

See also vim.type_idx and lua-special-tbl.

vim.types vim.types
Table with possible values for vim.type_idx. Contains two sets of key-value pairs: first maps possible values for vim.type_idx to human-readable strings, second maps human-readable type names to values for vim.type_idx. Currently contains pairs for `float`, `array` and `dictionary` types.

Note: One must expect that values corresponding to `vim.types.float`, `vim.types.array` and `vim.types.dictionary` fall under only two following assumptions:

1. Value may serve both as a key and as a value in a table. Given the properties of Lua tables this basically means “value is not `nil`”.
2. For each value in `vim.types` table `vim.types[vim.types[value]]` is the same as `value`. No other restrictions are put on types, and it is not guaranteed that values corresponding to `vim.types.float`, `vim.types.array` and `vim.types.dictionary` will not change or that `vim.types` table will only contain values for these three types.

log_levels vim.log.levels Log levels are one of the values defined in `vim.log.levels`:

vim.log.levels.DEBUG vim.log.levels.ERROR vim.log.levels.INFO vim.log.levels.TRACE vim.log.levels.WARN vim.log.levels.OFF

vim.empty_dict() vim.empty_dict()
Creates a special empty table (marked with a metatable), which Nvim converts to an empty dictionary when translating Lua values to Vimscript or API types. Nvim by default converts an empty table {} without this metatable to an list/array.

Note: If numeric keys are present in the table, Nvim ignores the metatable marker and converts the dict to a list/array anyway.

Return:

(table)

vim.iconv({str}, {from}, {to}) vim.iconv()
The result is a String, which is the text {str} converted from encoding {from} to encoding {to}. When the conversion fails nil is returned. When some characters could not be converted they are replaced with "?". The encoding names are whatever the iconv() library function can accept, see ":Man 3 iconv".

Parameters:
- {str} (string) Text to convert
- {from} (string) Encoding of {str}
- {to} (string) Target encoding

Return:
(string?) Converted string if conversion succeeds, nil otherwise.

vim.in_fast_event() vim.in_fast_event()
Returns true if the code is executing as part of a "fast" event handler, where most of the API is disabled. These are low-level events (e.g. lua-loop-callbacks) which can be invoked whenever Nvim polls for input. When this is false most API functions are callable (but may be subject to other restrictions such as textlock).

vim.rpcnotify({channel}, {method}, {...}) vim.rpcnotify()
Sends {event} to {channel} via RPC and returns immediately. If {channel} is 0, the event is broadcast to all channels.

This function also works in a fast callback lua-loop-callbacks.

Parameters:
- {channel} (integer)
- {method} (string)
- {...} (any?)

vim.rpcrequest({channel}, {method}, {...}) vim.rpcrequest()
Invokes RPC method on channel and blocks until a response is received.

Note: Msgpack NIL values in the response are represented as vim.NIL.

Example: see nvim_exec_lua()

Parameters:
- {channel} (integer)
- {method} (string)
- {...} (any?)

vim.schedule({fn}) vim.schedule()
Schedules {fn} to be invoked soon by the main event-loop. Useful to avoid textlock or other temporary restrictions.

Parameters:
- {fn} (fun())

Return (multiple):
(nil) result (string?) err Error message if scheduling failed, nil otherwise.

vim.str_utf_end({str}, {index}) vim.str_utf_end()
Gets the distance (in bytes) from the last byte of the codepoint (character) that {index} points to.

Examples:

```
-- The character 'æ' is stored as the bytes '\xc3\xa6' (using UTF-8)

-- Returns 0 because the index is pointing at the last byte of a character
vim.str_utf_end('æ', 2)

-- Returns 1 because the index is pointing at the penultimate byte of a character
vim.str_utf_end('æ', 1)
```

Parameters:
- {str} (string)
- {index} (integer)

Return:
(integer)

vim.str_utf_pos({str}) vim.str_utf_pos()
Gets a list of the starting byte positions of each UTF-8 codepoint in the given string.

Embedded NUL bytes are treated as terminating the string.

Parameters:
- {str} (string)

Return:
(integer[])

vim.str_utf_start({str}, {index}) vim.str_utf_start()
Gets the distance (in bytes) from the starting byte of the codepoint (character) that {index} points to.

The result can be added to {index} to get the starting byte of a character.

Examples:

```
-- The character 'æ' is stored as the bytes '\xc3\xa6' (using UTF-8)

-- Returns 0 because the index is pointing at the first byte of a character
vim.str_utf_start('æ', 1)

-- Returns -1 because the index is pointing at the second byte of a character
vim.str_utf_start('æ', 2)
```

Parameters:
- {str} (string)
- {index} (integer)

Return:
(integer)

vim.stricmp({a}, {b}) vim.stricmp()
Compares strings case-insensitively.

Parameters:
- {a} (string)
- {b} (string)

Return:
(0|1|-1) if strings are equal, {a} is greater than {b} or {a} is lesser than {b}, respectively.

vim.ui_attach({ns}, {opts}, {callback}) vim.ui_attach()
WARNING: This feature is experimental/unstable.

Subscribe to ui-events, similar to nvim_ui_attach() but receive events in a Lua callback. Used to implement screen elements like popupmenu or message handling in Lua.

{callback} receives event name plus additional parameters. See ui-popupmenu and the sections below for event format for respective events.

Callbacks for msg_show events originating from internal messages (as opposed to events from commands or API calls) are executed in api-fast context; showing the message needs to be scheduled.

Excessive errors inside the callback will result in forced detachment.

WARNING: This api is considered experimental. Usability will vary for different screen elements. In particular ext_messages behavior is subject to further changes and usability improvements. This is expected to be used to handle messages when setting 'cmdheight' to zero (which is likewise experimental).

Example (stub for a ui-popupmenu implementation):

```
ns = vim.api.nvim_create_namespace('my_fancy_pum')

vim.ui_attach(ns, {ext_popupmenu=true}, function(event, ...)
  if event == 'popupmenu_show' then
    local items, selected, row, col, grid = ...
    print('display pum ', #items)
  elseif event == 'popupmenu_select' then
    local selected = ...
    print('selected', selected)
  elseif event == 'popupmenu_hide' then
    print('FIN')
  end
end)
```

Parameters:
- {ns} (integer) Namespace ID
- {opts} (table<string, any>) Optional parameters.
  - {ext_…}? (boolean) Any of ui-ext-options, if true enable events for the respective UI element.
  - {set_cmdheight}? (boolean) If false, avoid setting 'cmdheight' to 0 when ext_messages is enabled.
- {callback} (fun(event: string, ...): any) Function called for each UI event. A truthy return value signals to Nvim that the event is handled, in which case it is not propagated to remote UIs.

vim.ui_detach({ns}) vim.ui_detach()
Detach a callback previously attached with vim.ui_attach() for the given namespace {ns}.

Parameters:
- {ns} (integer) Namespace ID

LUA-VIMSCRIPT BRIDGE lua-vimscript

Nvim Lua provides an interface or "bridge" to Vimscript variables and functions, and editor commands and options.

Objects passed over this bridge are COPIED (marshalled): there are no "references". lua-guide-variables For example, using vim.fn.remove() on a Lua list copies the list object to Vimscript and does NOT modify the Lua list:

```
local list = { 1, 2, 3 }
vim.fn.remove(list, 0)
vim.print(list)  --> "{ 1, 2, 3 }"
```

vim.call({func}, {...}) vim.call()
Invokes vim-function or user-function {func} with arguments {...}. See also vim.fn. Equivalent to:

```
vim.fn[func]({...})
```

vim.cmd({command}) See vim.cmd().

vim.fn.{func}({...}) vim.fn

Invokes vim-function or user-function {func} with arguments {...}. To call autoload functions, use the syntax:

```
vim.fn['some#function']({...})
```

Unlike vim.api.nvim_call_function() this converts directly between Vim objects and Lua objects. If the Vim function returns a float, it will be represented directly as a Lua number. Empty lists and dictionaries both are represented by an empty table.

Note: v:null values as part of the return value is represented as vim.NIL special value

Note: vim.fn keys are generated lazily, thus `pairs(vim.fn)` only enumerates functions that were called at least once.

Note: The majority of functions cannot run in api-fast callbacks with some undocumented exceptions which are allowed.

lua-vim-variables

The Vim editor global dictionaries g: w: b: t: v: can be accessed from Lua conveniently and idiomatically by referencing the `vim.*` Lua tables described below. In this way you can easily read and modify global Vimscript variables from Lua.

Example:

```
vim.g.foo = 5     -- Set the g:foo Vimscript variable.
print(vim.g.foo)  -- Get and print the g:foo Vimscript variable.
vim.g.foo = nil   -- Delete (:unlet) the Vimscript variable.
vim.b[2].foo = 6  -- Set b:foo for buffer 2
```

Note that setting dictionary fields directly will not write them back into Nvim. This is because the index into the namespace simply returns a copy. Instead the whole dictionary must be written as one. This can be achieved by creating a short-lived temporary.

Example:

```
vim.g.my_dict.field1 = 'value'  -- Does not work

local my_dict = vim.g.my_dict   --
my_dict.field1 = 'value'        -- Instead do
vim.g.my_dict = my_dict         --
```

vim.g vim.g

Global (g:) editor variables. Key with no value returns `nil`.

vim.b vim.b

Buffer-scoped (b:) variables for the current buffer. Invalid or unset key returns `nil`. Can be indexed with an integer to access variables for a specific buffer.

vim.w vim.w

Window-scoped (w:) variables for the current window. Invalid or unset key returns `nil`. Can be indexed with an integer to access variables for a specific window.

vim.t vim.t

Tabpage-scoped (t:) variables for the current tabpage. Invalid or unset key returns `nil`. Can be indexed with an integer to access variables for a specific tabpage.

vim.v vim.v

v: variables. Invalid or unset key returns `nil`.

lua-options
lua-vim-options
lua-vim-set
lua-vim-setlocal

Vim options can be accessed through vim.o, which behaves like Vimscript :set.

Examples:

To set a boolean toggle: Vimscript: `set number` Lua: `vim.o.number = true`

To set a string value: Vimscript: `set wildignore=*.o,*.a,__pycache__` Lua: `vim.o.wildignore = '*.o,*.a,__pycache__'` Lua (alt): `vim.o.wildignore = { '*.o', '*.a', '__pycache__' }`

To set a key:value type option: Vimscript: `set listchars=eol:~,space:-` Lua: `vim.o.listchars = 'eol:~,space:-'` Lua (alt): `vim.o.listchars = { eol = '~', space = '-' }`

Similarly, there is vim.bo and vim.wo for setting buffer-scoped and window-scoped options. Note that this must NOT be confused with local-options and :setlocal. There is also vim.go that only accesses the global value of a global-local option, see :setglobal.

vim.opt_local
vim.opt_global
vim.opt

A special interface vim.opt exists for conveniently interacting with list- and map-style options from Lua: It allows accessing them as Lua tables and offers object-oriented method for adding and removing entries.

Examples:

The following methods of setting a list-style option are equivalent: In Vimscript:

```
set wildignore=*.o,*.a,__pycache__
```

In Lua using `vim.o`:

```
vim.o.wildignore = '*.o,*.a,__pycache__'
```

In Lua using `vim.opt`:

```
vim.opt.wildignore = { '*.o', '*.a', '__pycache__' }
```

To replicate the behavior of :set+=, use:

```

vim.opt.wildignore:append { "*.pyc", "node_modules" }
```

To replicate the behavior of :set^=, use:

```

vim.opt.wildignore:prepend { "new_first_value" }
```

To replicate the behavior of :set-=, use:

```

vim.opt.wildignore:remove { "node_modules" }
```

The following methods of setting a map-style option are equivalent: In Vimscript:

```
set listchars=space:_,tab:>~
```

In Lua using `vim.o`:

```
vim.o.listchars = 'space:_,tab:>~'
```

In Lua using `vim.opt`:

```
vim.opt.listchars = { space = '_', tab = '>~' }
```

Note that vim.opt returns an `Option` object, not the value of the option, which is accessed through vim.opt:get():

Examples:

The following methods of getting a list-style option are equivalent: In Vimscript:

```
echo wildignore
```

In Lua using `vim.o`:

```
print(vim.o.wildignore)
```

In Lua using `vim.opt`:

```
vim.print(vim.opt.wildignore:get())
```

In any of the above examples, to replicate the behavior :setlocal, use `vim.opt_local`. Additionally, to replicate the behavior of :setglobal, use `vim.opt_global`.

Option:append({value}) vim.opt:append()
Append a value to string-style options. See :set+=

These are equivalent:

```
vim.opt.formatoptions:append('j')
vim.opt.formatoptions = vim.opt.formatoptions + 'j'
```

Parameters:

- {value} (`string`) Value to append

Option:get() vim.opt:get()
Returns a Lua-representation of the option. Boolean, number and string values will be returned in exactly the same fashion.

For values that are comma-separated lists, an array will be returned with the values as entries in the array:

```
vim.cmd [[set wildignore=*.pyc,*.o]]

vim.print(vim.opt.wildignore:get())
-- { "*.pyc", "*.o", }

for _, ignore_pattern in ipairs(vim.opt.wildignore:get()) do
    print("Will ignore:", ignore_pattern)
end
-- Will ignore: *.pyc
-- Will ignore: *.o
```

For values that are comma-separated maps, a table will be returned with the names as keys and the values as entries:

```
vim.cmd [[set listchars=space:_,tab:>~]]

vim.print(vim.opt.listchars:get())
--  { space = "_", tab = ">~", }

for char, representation in pairs(vim.opt.listchars:get()) do
    print(char, "=>", representation)
end
```

For values that are lists of flags, a set will be returned with the flags as keys and `true` as entries.

```
vim.cmd [[set formatoptions=njtcroql]]

vim.print(vim.opt.formatoptions:get())
-- { n = true, j = true, c = true, ... }

local format_opts = vim.opt.formatoptions:get()
if format_opts.j then
    print("J is enabled!")
end
```

Return:
(`string|integer|boolean|string[]|table<string,string|boolean>?`) value of option

Option:prepend({value}) vim.opt:prepend()
Prepend a value to string-style options. See :set^=

These are equivalent:

```
vim.opt.wildignore:prepend('*.o')
vim.opt.wildignore = vim.opt.wildignore ^ '*.o'
```

Parameters:
- {value} (`string`) Value to prepend

Option:remove({value}) vim.opt:remove()
Remove a value from string-style options. See :set-=

These are equivalent:

```
vim.opt.wildignore:remove('*.pyc')
vim.opt.wildignore = vim.opt.wildignore - '*.pyc'
```

Parameters:
- {value} (`string`) Value to remove

vim.bo[{bufnr}] vim.bo
Gets or sets buffer-scoped options on buffer {bufnr} (or "current buffer" if 0 or omitted). Like `:setlocal`. Invalid {bufnr} or key is an error.

Example:

```
local bufnr = vim.api.nvim_get_current_buf()
vim.bo[bufnr].buflisted = true    -- same as vim.bo.buflisted = true
print(vim.bo.comments)
print(vim.bo.baz)                 -- error: invalid key
```

vim.env vim.env
Gets or sets environment variables in the current editor process. See expand-env and :let-environment for the Vimscript behavior. Invalid or unset key returns `nil`.

Example:

```
vim.env.FOO = 'bar'
print(vim.env.TERM)
```

vim.go vim.go
Gets or sets global options. Like `:setglobal`. Invalid key is an error.

Note: unlike vim.o, this accesses the global option value and thus is mostly useful with global-local options.

Example:

```
vim.go.cmdheight = 4
print(vim.go.columns)
print(vim.go.bar)     -- error: invalid key
```

vim.o vim.o
Gets or sets options. Works like `:set`, so buffer/window-scoped options target the current buffer/window. Invalid key is an error.

Example:

```
vim.o.cmdheight = 4
print(vim.o.columns)
print(vim.o.foo)     -- error: invalid key
```

vim.wo[{winid}][{bufnr}] vim.wo
Gets or sets window-scoped options on window {winid} (or "current window" if 0 or omitted) and buffer {bufnr} (0 for current buffer). Like `:setlocal` if setting a global-local option or if {bufnr} is specified, like `:set` otherwise. Invalid {winid}, {bufnr}, or key is an error.

Note: only bufnr=0 (current window-buffer) is supported, currently.

Example:

```
local winid = vim.api.nvim_get_current_win()
vim.wo[winid].number = true    -- same as vim.wo.number = true
print(vim.wo.foldmarker)
print(vim.wo.quux)             -- error: invalid key
vim.wo[winid][0].spell = false -- like ':setlocal nospell'
```

Lua module: vim

vim.Ringbuf

Fields:

- {clear} (fun()) See Ringbuf:clear().
- {peek} (fun(): T?) See Ringbuf:peek().
- {pop} (fun(): T?) See Ringbuf:pop().
- {push} (fun(item: T)) See Ringbuf:push().

Ringbuf:clear() Ringbuf:clear()
Clear all items

Ringbuf:peek() Ringbuf:peek()
Returns the first unread item without removing it

Return:
(T?)

Ringbuf:pop() Ringbuf:pop()
Removes and returns the first unread item

Return:
(T?)

Ringbuf:push({item}) Ringbuf:push()
Adds an item, overriding the oldest item if the buffer is full.

Parameters:

- {item} (T)

vim.cmd({cmd}) vim.cmd()
Executes Vimscript (Ex-commands).

Can be indexed with a command name to get a function, thus you can write vim.cmd.echo(…) instead of vim.cmd{cmd='echo',…}.

Examples:

```
-- Single command:
vim.cmd('echo 42')
-- Multiline script:
vim.cmd([[
  augroup my.group
    autocmd!
    autocmd FileType c setlocal cindent
  augroup END
]])

-- Ex command :echo "foo". Note: string literals must be double-quoted.
vim.cmd('echo "foo"')
vim.cmd { cmd = 'echo', args = { '"foo"' } }
vim.cmd.echo({ args = { '"foo"' } })
vim.cmd.echo('"foo"')

-- Ex command :write! myfile.txt
vim.cmd('write! myfile.txt')
vim.cmd { cmd = 'write', args = { 'myfile.txt' }, bang = true }
vim.cmd.write { args = { 'myfile.txt' }, bang = true }
vim.cmd.write { 'myfile.txt', bang = true }

-- Ex command :vertical resize +2
vim.cmd.resize({ '+2', mods = { vertical = true } })

-- Pass arg literally, without needing to escape special chars:
vim.cmd.edit({ '%foo"|bar#baz"', magic = { file = false, bar = false } })
```

Parameters:

- {cmd} (string|table) Command(s) to execute.
  - The string form supports multiline Vimscript (alias to nvim_exec2(), behaves like :source).
  - The table form executes a single command (alias to nvim_cmd()).

See also:

- ex-cmd-index

vim.deep_equal({a}, {b}) vim.deep_equal()
Deep compare values for equality

Tables are compared recursively unless they both provide the eq metamethod. All other types are compared using the equality == operator. Cyclic tables are supported.

Parameters:

- {a} (any) First value

- {b} (any) Second value

Return:
(boolean) `true` if values are equals, else `false`

vim.deepcopy({orig}, {noref}) vim.deepcopy()
Returns a deep copy of the given object. Non-table objects are copied as in a typical Lua assignment, whereas table objects are copied recursively. Functions are naively copied, so functions in the copied table point to the same functions as those in the input table. Userdata and threads are not copied and will throw an error.

Note: `noref=true` is much more performant on tables with unique table fields, while `noref=false` is more performant on tables that reuse table fields.

Parameters:
- {orig} (table) Table to copy
- {noref} (boolean?) When `false` (default) a contained table is only copied once and all references point to this single copy. When `true` every occurrence of a table results in a new copy. This also means that a cyclic reference can cause `deepcopy()` to fail.

Return:
(table) Table of copied keys and (nested) values.

vim.defaulttable({createfn}) vim.defaulttable()
Creates a table whose missing keys are provided by {createfn} (like Python's "defaultdict").

If {createfn} is `nil` it defaults to defaulttable() itself, so accessing nested keys creates nested tables:

```
local a = vim.defaulttable()
a.b.c = 1
```

Parameters:
- {createfn} (fun(key:any):any?) Provides the value for a missing key.

Return:
(table) Empty table with `__index` metamethod.

vim.defer_fn({fn}, {timeout}) vim.defer_fn()
Defers calling {fn} until {timeout} ms passes.

Use to do a one-shot timer that calls {fn}. **Note:** The {fn} is scheduled automatically, so API functions are safe to call.

Parameters:
- {fn} (function) Callback to call once `timeout` expires
- {timeout} (integer) Number of milliseconds to wait before calling `fn`

Return:
(uv.uv_timer_t) timer luv timer object

vim.deprecate() vim.deprecate({name}, {alternative}, {version}, {plugin}, {backtrace}) Shows a deprecation message to the user.

Parameters:
- {name} (string) **Deprecated** feature (function, API, etc.).
- {alternative} (string?) Suggested alternative feature.
- {version} (string) Version when the deprecated function will be removed.
- {plugin} (string?) Name of the plugin that owns the deprecated feature. Defaults to "Nvim".
- {backtrace} (boolean?) Prints backtrace. Defaults to true.

Return:
(string?) Deprecation message, or nil if no message was shown.

vim.endswith({s}, {suffix}) vim.endswith()
Tests if s ends with `suffix`.

Parameters:
- {s} (string) String
- {suffix} (string) Suffix to match

Return:
(boolean) `true` if `suffix` is a suffix of s

vim.gsplit({s}, {sep}, {opts}) vim.gsplit()
Gets an iterator that splits a string at each instance of a separator, in "lazy" fashion (as opposed to vim.split() which is "eager").

Example:

```
for s in vim.gsplit(':aa::b:', ':', {plain=true}) do
  print(s)
end
```

If you want to also inspect the separator itself (instead of discarding it), use string.gmatch(). Example:

```
for word, num in ('foo111bar222'):gmatch('([^0-9]*)(%d*)') do
  print(('word: %s num: %s'):format(word, num))
end
```

Parameters:
- {s} (string) String to split
- {sep} (string) Separator or pattern
- {opts} (table?) Keyword arguments kwargs:
  - {plain}? (boolean) Use sep literally (as in string.find).

- `{trimempty}`? (`boolean`) Discard empty segments at start and end of the sequence.

Return:
(`fun():string`?) Iterator over the split components

See also:
- string.gmatch()
- vim.split()
- lua-patterns
- https://www.lua.org/pil/20.2.html
- http://lua-users.org/wiki/StringLibraryTutorial

vim.inspect({`root`}, {`options`}) vim.inspect()
Gets a human-readable representation of the given object.

Parameters:
- `{root}` (`any`)
- `{options}` (`table`?) A table with the following fields:
  - `{depth}`? (`integer`, default: unlimited) Maximum table nesting depth. Deeper tables are shown as `{...}`.
  - `{indent}`? (`string`, default: `"  "`) String repeated for each level of indentation.
  - `{newline}`? (`string`, default: `"\n"`) String used for line breaks.
  - `{process}`? (`fun(item: any, path: any[]): any`) Transforms values, keys, and metatables before formatting. Return a replacement value, or `nil` to omit a table entry. `{path}` lists the keys leading to `{item}`, with `vim.inspect.KEY` appended for keys or `vim.inspect.METATABLE` for metatables. The root has an empty path.

Return:
(`string`)

See also:
- vim.print()
- https://github.com/kikito/inspect.lua
- https://github.com/mpeterv/vinspect

vim.is_callable({`f`}) vim.is_callable()
Returns true if object `f` can be called as a function.

Parameters:
- `{f}` (`any`?) Any object

Return:
(`boolean`) `true` if `f` is callable, else `false`

vim.isarray({`t`}) vim.isarray()
Tests if `t` is an "array": a table indexed only by integers (potentially non-contiguous).

If the indexes start from 1 and are contiguous then the array is also a list. vim.islist()

Empty table {} is an array, unless it was created by vim.empty_dict() or returned as a dict-like API or Vimscript result, for example from rpcrequest() or vim.fn.

Parameters:
- `{t}` (`any`?)

Return:
(`boolean`) `true` if array-like table, else `false`.

See also:
- https://github.com/openresty/luajit2#tableisarray

vim.islist({`t`}) vim.islist()
Tests if `t` is a "list": a table indexed only by contiguous integers starting from 1 (what lua-length calls a "regular array").

Empty table {} is a list, unless it was created by vim.empty_dict() or returned as a dict-like API or Vimscript result, for example from rpcrequest() or vim.fn.

Parameters:
- `{t}` (`any`?)

Return:
(`boolean`) `true` if list-like table, else `false`.

See also:
- vim.isarray()

vim.isnil({`t`}) vim.isnil()
Tests if `t` is `nil` or vim.NIL.

Attributes:
Since: 0.13.0

Parameters:
- `{t}` (`any`?)

Return:
(`boolean`) `true` if `nil` or vim.NIL, else `false`.

vim.keycode({`keys`}, {`info`}) vim.keycode()
Converts keys from key-notation to the internal encoding. Optionally returns structured key-chord info as retval 2.

Inverse of keytrans(), which converts the internal encoding back to key-notation.

Examples:

```
local k = vim.keycode
vim.g.mapleader = k'<bs>'

-- Split a key sequence into chords, e.g. to inspect modifiers.
local _, chords = vim.keycode('<C-w>v', true)
assert(chords[1].key == 'w' and chords[1].mod[1] == 'C')

-- keytrans() is the inverse: internal encoding => key-notation.
assert(vim.fn.keytrans(vim.keycode('<C-a>')) == '<C-A>')
```

Parameters:
- {keys} (string) Keys in key-notation.
- {info} (boolean?) Also return key-chord info.

Return (multiple):
(string) Internal representation of the given keys. (table?) List of parsed key-chords, each with fields:
- {key} (string) Key without modifiers. Example: <C-A> has key="a".
- {key_alt} (string?) Alternative spelling of key, or nil if There Is No Alternative (TINA). Example: "<" has key_alt="lt".
- {keys} (string) Full key-chord in canonical key-notation (as produced by keytrans()), including modifiers. Example: <A-f> has keys="<M-f>".
- {mod} (('M'|'T'|'C'|'S'|'2'|'3'|'4'|'D')[]) List of key-chord modifiers. Example: <C-S-f> has mod={ 'C', 'S' }.

See also:
- nvim_replace_termcodes()
- keytrans()

vim.list.bisect({t}, {val}, {opts}) vim.list.bisect()
Search for a position in a sorted lua-list {t} where {val} can be inserted while keeping the list sorted.

Use {bound} to determine whether to return the first or the last position, defaults to "lower", i.e., the first position.

With {opts.key}, {val} may differ from the list elements, provided the key function accepts both. For a string key, that field must exist on both. A partial record is sufficient if it contains all fields used by the key.

NOTE: The values being compared must support <, and the list must be sorted by those values (after applying {opts.key}, if provided).

Example:

```
local t = { 1, 2, 2, 3, 3, 3 }
local first = vim.list.bisect(t, 3)
-- `first` is `val`'s first index if found,
-- useful for existence checks.
print(t[first]) -- 3

local last = vim.list.bisect(t, 3, { bound = 'upper' })
-- Note that `last` is 7, not 6,
-- this is suitable for insertion.

table.insert(t, last, 4)
-- t is now { 1, 2, 2, 3, 3, 3, 4 }

-- You can use lower bound and upper bound together
-- to obtain the range of occurrences of `val`.

-- 3 is in [first, last)
for i = first, last - 1 do
  print(t[i]) -- { 3, 3, 3 }
end
```

Attributes:
Since: 0.12.0

Parameters:
- {t} (any[]) A sorted list.
- {val} (any) The value to search.
- {opts} (table?) A table with the following fields:
  - {bound}? ('lower'|'upper', default: 'lower') Specifies the search variant.
    - "lower": returns the first position where inserting {val} keeps the list sorted.
    - "upper": returns the last position where inserting {val} keeps the list sorted..
  - {hi}? (integer, default: #t + 1) End index of the list, exclusive.
  - {key}? ((string & keyof T)|fun(val: T): any) Applied to {val} and the list elements being compared. If a string, index both by this field name. If a function, it must accept both {val} and the list elements and return mutually comparable keys.
  - {lo}? (integer, default: 1) Start index of the list.

Overloads:
- fun<T, Q>(t: T[], val: Q, opts: vim.list.bisect.Opts<T|Q> & { key: fun(val: T|Q): any }): integer
- fun<T: table, Q: table>(t: T[], val: Q, opts: vim.list.bisect.Opts<T|Q> & { key: string & keyof T & keyof Q }): integer

Return:
(integer) index serves as either the lower bound or the upper bound position.

vim.list.unique({t}, {key}) vim.list.unique()
Removes duplicate values from a lua-list in-place.

Only the first occurrence of each value is kept. The operation is performed in-place and the input table is modified.

Accepts an optional key argument, which if provided is called for each value in the list to compute a hash key for uniqueness comparison. If key is a string, it is used as the field name to index each value. This is useful for deduplicating table values or complex objects. If key returns `nil` for a value, that value will be considered unique, even if multiple values return `nil`.

Example:

```
local t = {1, 2, 2, 3, 1}
vim.list.unique(t)
-- t is now {1, 2, 3}

local t = { {id=1}, {id=2}, {id=1} }
vim.list.unique(t, 'id')
-- t is now { {id=1}, {id=2} }
```

Attributes:
Since: 0.12.0

Parameters:
- `{t}` (`any[]`)
- `{key}` (`string|fun(x: T): any?`) Optional field name or hash function to determine uniqueness of values

Return:
(`any[]`) The deduplicated list

See also:
- Iter:unique()

vim.list_contains(`{t}`, `{value}`) vim.list_contains()
Checks if a list-like table (integer keys without gaps) contains `value`.

Parameters:
- `{t}` (`table`) Table to check (must be list-like, not validated)
- `{value}` (any) Value to compare

Return:
(`boolean`) `true` if `t` contains `value`

See also:
- vim.tbl_contains() for checking values in general tables

vim.list_extend(`{dst}`, `{src}`, `{start}`, `{finish}`) vim.list_extend()
Extends a list-like table with the values of another list-like table.

NOTE: This mutates dst!

Parameters:
- `{dst}` (`table`) List which will be modified and appended to
- `{src}` (`table`) List from which values will be inserted
- `{start}` (`integer?`) (default: 1) Start index on src.
- `{finish}` (`integer?`) (default: `#src`) Final index on src.

Return:
(`table`) dst

See also:
- vim.tbl_extend()

vim.list_slice(`{list}`, `{start}`, `{finish}`) vim.list_slice()
Creates a copy of a table containing only elements from start to end (inclusive)

Parameters:
- `{list}` (`any[]`) Table
- `{start}` (`integer?`) Start range of slice
- `{finish}` (`integer?`) End range of slice

Return:
(`any[]`) Copy of table sliced from start to finish (inclusive)

vim.lua_omnifunc(`{find_start}`) vim.lua_omnifunc()
Omnifunc for completing Lua values from the runtime Lua interpreter, similar to the builtin completion for the `:lua` command.

Activate using `vim.bo.omnifunc = vim.lua_omnifunc` in a Lua buffer.

Parameters:
- `{find_start}` (`1|0`)

vim.nonnil(`{...}`) vim.nonnil()
Returns the first argument which is not nil.

If all arguments are nil, returns nil.

Example:

```
local a = nil
local b = nil
local c = 42
local d = true
assert(vim.nonnil(a, b, c, d) == 42)
```

Attributes:
Since: 0.13.0

Parameters:
- `{...}` (any)

Return:
(any)

vim.notify({msg}, {level}, {opts}) vim.notify()
Displays a notification to the user.

This function can be overridden by plugins to display notifications using a custom provider (such as the system notification provider). By default, writes to :messages.

Parameters:
- `{msg}` (`string`) Content of the notification to show to the user.
- `{level}` (`integer?`) One of the values from vim.log.levels.
- `{opts}` (`table?`) Optional parameters. Unused by default.

vim.notify_once({msg}, {level}, {opts}) vim.notify_once()
Displays a notification only one time.

Like vim.notify(), but subsequent calls with the same message will not display a notification.

Parameters:
- `{msg}` (`string`) Content of the notification to show to the user.
- `{level}` (`integer?`) One of the values from vim.log.levels.
- `{opts}` (`table?`) Optional parameters. Unused by default.

Return:
(`boolean`) true if message was displayed, else false

vim.npcall({fn}, {...}) vim.npcall()
Calls the function `fn` in `protected mode` like pcall(), but returns `nil` on error.

Attributes:
Since: 0.13.0

Parameters:
- `{fn}` (`fun(...):T`)
- `{...}` (`any?`)

Return:
(any) ...

vim.on_key({fn}, {ns_id}, {opts}) vim.on_key()
Registers function `{fn}` with namespace `{ns_id}` as a listener to EVERY input key (except keys consumed by getchar(), by `{fn}` itself, or by a `<Cmd>` payload).

To parse key-chords, see vim.keycode(). Example:

```
local keychords = vim.keycode(vim.fn.keytrans(key), true)
```

Note:
- If `{fn}` returns an empty string, `{key}` is discarded/ignored; if `{key}` is `<Cmd>` then the "`<Cmd>`…`<CR>`" sequence is discarded as a whole.
- Non-recursive: if `{fn}` itself consumes input, it won't be invoked for those keys.
- Not invoked for keys consumed by getchar()/getcharstr(): they never reach the main loop.
- To UNregister a given `{ns_id}`, pass `nil` `{fn}`.
- `{fn}` will be removed on error.
- `{fn}` will not be cleared by nvim_buf_clear_namespace()
- The -w command-line option is related but does not support callbacks and cannot be toggled dynamically.

Parameters:
- `{fn}` (`fun(key: string, typed: string): string??`) Function invoked for every input key, after mappings have been applied but before further processing. Arguments `{key}` and `{typed}` are raw input (use keytrans() to get keycodes).
  - `{key}` is the key after mappings are applied.
  - `{typed}` is the input before mappings are applied; may be empty if `{key}` was produced by non-typed key(s) or by the same typed key(s) that produced a previous `{key}`.
- `{ns_id}` (`integer?`) Namespace ID. If nil or 0, returns a new namespace id.
- `{opts}` (`table?`) Optional parameters

Return:
(`integer`) Namespace id associated with `{fn}`. Or count of all callbacks if on_key() is called without arguments.

See also:
- keytrans()
- vim.keycode()

vim.paste({lines}, {phase}) vim.paste()
Paste handler, invoked by nvim_paste().

Note: This is provided only as a "hook", don't call it directly; call nvim_paste() instead, which arranges redo (dot-repeat) and invokes `vim.paste`.

Example: To remove ANSI color codes when pasting:

```
vim.paste = (function(overridden)
  return function(lines, phase)
    for i,line in ipairs(lines) do
      -- Scrub ANSI color codes from paste input.
      lines[i] = line:gsub('\27%[[0-9;mK]+', '')
    end
    return overridden(lines, phase)
  end
end)(vim.paste)
```

Parameters:
- {lines} (string[]) readfile()-style list of lines to paste. channel-lines
- {phase} (-1|1|2|3) -1: "non-streaming" paste: the call contains all lines. If paste is "streamed", phase indicates the stream state:
  - 1: starts the paste (exactly once)
  - 2: continues the paste (zero or more times)
  - 3: ends the paste (exactly once)

Return:
(boolean) result false if client should cancel the paste.

See also:
- paste

vim.pesc({s}) vim.pesc()
Escapes magic chars in lua-patterns.

Parameters:
- {s} (string) String to escape

Return:
(string) %-escaped pattern string

See also:
- https://github.com/rxi/lume

vim.print({...}) vim.print()
"Pretty prints" the given arguments and returns them unmodified.

Example:

```
local hl_normal = vim.print(vim.api.nvim_get_hl(0, { name = 'Normal' }))
```

Parameters:
- {...} (any)

Return:
(any) given arguments.

See also:
- vim.inspect()
- :=

vim.ringbuf({size}) vim.ringbuf()
Create a ring buffer limited to a maximal number of items. Once the buffer is full, adding a new entry overrides the oldest entry.

```
local ringbuf = vim.ringbuf(4)
ringbuf:push("a")
ringbuf:push("b")
ringbuf:push("c")
ringbuf:push("d")
ringbuf:push("e")    -- overrides "a"
print(ringbuf:pop()) -- returns "b"
print(ringbuf:pop()) -- returns "c"

-- Can be used as iterator. Pops remaining items:
for val in ringbuf do
  print(val)
end
```

Returns a Ringbuf instance with the following methods:
- Ringbuf:push()
- Ringbuf:pop()
- Ringbuf:peek()
- Ringbuf:clear()

Parameters:
- {size} (integer)

Return:
(vim.Ringbuf<any>) ringbuf

vim.schedule_wrap({fn}) vim.schedule_wrap()
Returns a function which calls {fn} via vim.schedule().

The returned function passes all arguments to {fn}.

Example:

```
function notify_readable(_err, readable)
  vim.notify("readable? " .. tostring(readable))
end
vim.uv.fs_access(vim.fn.stdpath("config"), "R", vim.schedule_wrap(notify_readable))
```

Parameters:
- {fn} (function)

Return:
(function)

See also:
- lua-loop-callbacks
- vim.schedule()
- vim.in_fast_event()

vim.spairs({t}) vim.spairs()
Enumerates key-value pairs of a table, ordered by key.

Parameters:
- {t} (table) Dict-like table

Return (multiple):
(fun(table: table<K, V>, index?: K):K, V) for-in iterator over sorted keys and their values (table)

See also:
- Based on https://github.com/premake/premake-core/blob/master/src/base/table.lua

vim.split({s}, {sep}, {opts}) vim.split()
Splits a string at each instance of a separator and returns the result as a table (unlike vim.gsplit()).

Examples:

```
split(":aa::b:", ":")                   --> {'','aa','','b',''}
split("axaby", "ab?")                   --> {'','x','y'}
split("x*yz*o", "*", {plain=true})      --> {'x','yz','o'}
split("|x|y|z|", "|", {trimempty=true}) --> {'x', 'y', 'z'}
```

Parameters:
- {s} (string) String to split
- {sep} (string) Separator or pattern
- {opts} (table?) Keyword arguments kwargs:
  - {plain}? (boolean) Use sep literally (as in string.find).
  - {trimempty}? (boolean) Discard empty segments at start and end of the sequence.

Return:
(string[]) List of split components

See also:
- vim.gsplit()
- string.gmatch()

vim.startswith({s}, {prefix}) vim.startswith()
Tests if s starts with prefix.

Parameters:
- {s} (string) String
- {prefix} (string) Prefix to match

Return:
(boolean) true if prefix is a prefix of s

vim.str_byteindex()
vim.str_byteindex({s}, {encoding}, {index}, {strict_indexing}) Convert UTF-32, UTF-16 or UTF-8 {index} to byte index. If {strict_indexing} is false then an out of range index will return byte length instead of throwing an error.

Invalid UTF-8 and NUL is treated like in vim.str_utfindex(). An {index} in the middle of a UTF-16 sequence is rounded upwards to the end of that sequence.

Parameters:
- {s} (string)
- {encoding} ("utf-8"|"utf-16"|"utf-32")
- {index} (integer)
- {strict_indexing} (boolean?) default: true

Return:
(integer)

vim.str_utfindex()
vim.str_utfindex({s}, {encoding}, {index}, {strict_indexing}) Convert byte index to UTF-32, UTF-16 or UTF-8 indices. If {index} is not supplied, the length of the string is used. All indices are zero-based.

If {strict_indexing} is false then an out of range index will return string length instead of throwing an error. Invalid UTF-8 bytes, and embedded surrogates are counted as one code point each. An {index} in the middle of a UTF-8 sequence is rounded upwards to the end of that sequence.

Parameters:
- {s} (string)
- {encoding} ("utf-8"|"utf-16"|"utf-32")
- {index} (integer?)

- {strict_indexing} (boolean?) default: true

Return:
(integer)

vim.tbl_contains({t}, {value}, {opts}) vim.tbl_contains()
Checks if a table contains a given value, specified either directly or via a predicate that is checked for each value.

Example:

```
vim.tbl_contains({ 'a', { 'b', 'c' } }, function(v)
  return vim.deep_equal(v, { 'b', 'c' })
end, { predicate = true })
-- true
```

Parameters:
- {t} (table) Table to check
- {value} (any) Value to compare or predicate function reference
- {opts} (table?) Keyword arguments kwargs:
  - {predicate}? (boolean, default: false) value is a function reference to be checked

Return:
(boolean) true if t contains value

See also:
- vim.list_contains() for checking values in list-like tables

vim.tbl_count({t}) vim.tbl_count()
Counts the number of non-nil values in table t.

```
vim.tbl_count({ a=1, b=2 })  --> 2
vim.tbl_count({ 1, 2 })      --> 2
```

Parameters:
- {t} (table) Table

Return:
(integer) Number of non-nil values in table

See also:
- https://github.com/Tieske/Penlight/blob/master/lua/pl/tablex.lua

vim.tbl_deep_extend({behavior}, {...}) vim.tbl_deep_extend()
Merges two or more tables recursively.

Only lua-dict tables are merged recursively; lua-list tables are treated as opaque values (overwritten instead of merged). That is convenient for merging default/user configurations where lists typically should not be merged together.

Example:

```
-- Set `config.settings.…` without worrying about whether intermediate dicts exist.
local config = { settings = { tailwindCSS = { foo = 'bar' } } }
local merged = vim.tbl_deep_extend('force',
  config,
  { settings = { tailwindCSS = { experimental = { configFile = '/my/config.json' } } } }
)
vim.print(merged)
```

Parameters:
- {behavior} ('error'|'keep'|'force'|fun(key:any, prev_value:any?, value:any): any) Decides what to do if a key is found in more than one map:
  - "error": raise an error
  - "force": use value from the rightmost map
  - "keep": use value from the leftmost map
  - If a function, it receives the current key, the previous value in the currently merged table (if present), the current value and should return the value for the given key in the merged table.
- {...} (table) Two or more tables

Return:
(table) Merged table

See also:
- vim.tbl_extend()

vim.tbl_extend({behavior}, {...}) vim.tbl_extend()
Merges two or more tables.

Parameters:
- {behavior} ('error'|'keep'|'force'|fun(key:any, prev_value:any?, value:any): any) Decides what to do if a key is found in more than one map:
  - "error": raise an error
  - "force": use value from the rightmost map
  - "keep": use value from the leftmost map
  - If a function, it receives the current key, the previous value in the currently merged table (if present), the current value and should return the value for the given key in the merged table.
- {...} (table) Two or more tables

Return:
(table) Merged table

See also:
- extend()

vim.tbl_filter({fn}, {t}) vim.tbl_filter()
Filter a table using a predicate function

Parameters:
- {fn} (function) Function
- {t} (table) Table

Return:
(any[]) Table of filtered values

vim.tbl_get({o}, {...}) vim.tbl_get()
Gets a (nested) value from table o given by a sequence of keys ..., or nil if not found.

Note: To set deeply nested keys, see vim.tbl_deep_extend().

Example:

```
local o = { a = { b = true } }
-- Get `o.a.b`.
vim.print(vim.tbl_get(o, 'a', 'b')) -- true
o.a = {}
vim.print(vim.tbl_get(o, 'a', 'b')) -- nil
```

Parameters:
- {o} (table) Table to index
- {...} (any) Optional keys (0 or more, variadic) via which to index the table

Return:
(any) Nested value indexed by key (if it exists), else nil

See also:
- unpack()
- vim.tbl_deep_extend()

vim.tbl_isempty({t}) vim.tbl_isempty()
Checks if a table is empty.

Parameters:
- {t} (table) Table to check

Return:
(boolean) true if t is empty

See also:
- https://github.com/premake/premake-core/blob/master/src/base/table.lua

vim.tbl_keys({t}) vim.tbl_keys()
Return a list of all keys used in a table. However, the order of the return table of keys is not guaranteed.

Parameters:
- {t} (table) Table

Return:
(any[]) List of keys

See also:
- From https://github.com/premake/premake-core/blob/master/src/base/table.lua

vim.tbl_map({fn}, {t}) vim.tbl_map()
Applies function fn to all values of table t, in pairs() iteration order (which is not guaranteed to be stable, even when the data doesn't change).

Parameters:
- {fn} (fun(value: T): R) Function
- {t} (table<any, T>) Table

Return:
(table<any, R>) Table of transformed values

vim.tbl_values({t}) vim.tbl_values()
Return a list of all values used in a table. However, the order of the return table of values is not guaranteed.

Parameters:
- {t} (table) Table

Return:
(any[]) List of values

vim.trim({s}) vim.trim()
Trim whitespace (Lua pattern "%s") from both sides of a string.

Parameters:
- {s} (string) String to trim

Return:
(string) String with whitespace removed from its beginning and end

See also:
- lua-patterns
- https://www.lua.org/pil/20.2.html

vim.validate() vim.validate({name}, {value}, {validator}, {optional}, {message}) Validate function arguments.

This function has two valid forms:

1. `vim.validate(name, value, validator[, optional][, message])` Validates that argument {name} with value {value} satisfies {validator}. If {optional} is given and is `true`, then {value} may be `nil`. If {message} is given, then it is used as the expected type in the error message. Example:

```
 function vim.startswith(s, prefix)
  vim.validate('s', s, 'string')
  vim.validate('prefix', prefix, 'string')
  -- ...
end
```

2. `vim.validate(spec)` (DEPRECATED) where spec is of type `table<string,[value:any, validator: vim.validate.Validator, optional_or_msg? : boolean|string]>`) Validates a argument specification. Specs are evaluated in alphanumeric order, until the first failure.

Examples:

```
vim.validate('arg1', {'foo'}, 'table')
   --> NOP (success)
vim.validate('arg2', 'foo', 'string')
   --> NOP (success)

vim.validate('arg1', 1, 'table')
   --> error('arg1: expected table, got number')

vim.validate('arg1', 3, function(a) return (a % 2) == 0 end, 'even number')
   --> error('arg1: expected even number, got 3')

-- If multiple types are valid they can be given as a list:
vim.validate('arg1', {'foo'}, {'table', 'string'})
vim.validate('arg2', 'foo', {'table', 'string'})
-- NOP (success)
vim.validate('arg1', 1, {'string', 'table'})
-- error('arg1: expected string|table, got number')
```

Note:

- `validator` set to a value returned by lua-type() provides the best performance.

Parameters:

- {name} (`string`) Argument name
- {value} (`any`) Argument value
- {validator} (`vim.validate.Validator`)
  - (`string|string[]`): Any value that can be returned from lua-type() in addition to `'callable'`: `'boolean'`, `'callable'`, `'function'`, `'nil'`, `'number'`, `'string'`, `'table'`, `'thread'`, `'userdata'`.
  - (`fun(val:any): boolean, string?`) A function that returns a boolean and an optional string message.
- {optional} (`boolean?`) (default: false) Parameter is optional (may be omitted or nil)
- {message} (`string?`) message when validation fails

Overloads:

- `fun(name: string, val: any, validator: vim.validate.Validator, message: string)`
- `fun(spec: table<string,[any, vim.validate.Validator, boolean|string]>)`

vim.wait({time}, {callback}, {interval}, {fast_only}) vim.wait()
Waits up to `time` milliseconds, until `callback` returns `true` (success). Executes `callback` immediately, then on user events, internal events, and approximately every `interval` milliseconds. Returns `true` plus any remaining callback results on success.

Nvim processes other events while waiting. Cannot be called during an api-fast event.

Examples:

```
-- Wait for 100 ms, allowing other events to process.
vim.wait(100)

-- Wait up to 1000 ms or until `vim.g.foo` is true, at intervals of ~500 ms.
vim.wait(1000, function() return vim.g.foo end, 500)

-- Wait indefinitely until `vim.g.foo` is true, and get the callback results.
local ok, rv1, rv2, rv3 = vim.wait(math.huge, function()
  return vim.g.foo, 'a', 42, { ok = { 'yes' } }
end)

-- Schedule a function to set a value in 100ms. This would wait 10s if blocked, but actually
-- only waits 100ms because `vim.wait` processes other events while waiting.
vim.defer_fn(function() vim.g.timer_result = true end, 100)
if vim.wait(10000, function() return vim.g.timer_result end) then
  print('Only waiting a little bit of time!')
end

-- Yield via vim.wait() to allow CTRL-C to interrupt Lua code.
while true do
  -- ...work...
  local _, code = vim.wait(0, nil, 0)
  if code == -2 then -- CTRL-C.
    break
  end
end
```

Parameters:

- {time} (number) Number of milliseconds to wait. Must be non-negative number, any fractional part is truncated.
- {callback} (fun(): boolean, ...?) Optional callback. Waits until {callback} returns true
- {interval} (integer?) (default: 200) Milliseconds (approximate) to wait between polls.
- {fast_only} (boolean?) If true, only api-fast events will be processed.

Return (multiple):
(boolean) (-1|-2?)

- If callback returns true before timeout: true, ... (remaining callback results).
- On timeout: false, -1
- On interrupt: false, -2
- On error: the error is raised.

Lua module: vim.inspector

vim.inspector

vim.inspect_pos({buf}, {row}, {col}, {filter}) vim.inspect_pos()
Get all the items at a given buffer position.

Can also be pretty-printed with :Inspect!. :Inspect!

Attributes:
Since: 0.9.0

Parameters:

- {buf} (integer?) defaults to the current buffer
- {row} (integer?) row to inspect, 0-based. Defaults to the row of the current cursor
- {col} (integer?) col to inspect, 0-based. Defaults to the col of the current cursor
- {filter} (table?) Table with key-value pairs to filter the items
  - {extmarks} (boolean|"all", default: true) Include extmarks. When all, then extmarks without a hl_group will also be included.
  - {semantic_tokens} (boolean, default: true) Include semantic token highlights.
  - {syntax} (boolean, default: true) Include syntax based highlight groups.
  - {treesitter} (boolean, default: true) Include treesitter based highlight groups.

Return:
(table) a table with the following key-value pairs. Items are in "traversal order":

- treesitter: a list of treesitter captures
- syntax: a list of syntax groups
- semantic_tokens: a list of semantic tokens
- extmarks: a list of extmarks
- buffer: the buffer used to get the items
- row: the row used to get the items
- col: the col used to get the items

vim.show_pos({buf}, {row}, {col}, {filter}) vim.show_pos()
Show all the items at a given buffer position.

Can also be shown with :Inspect. :Inspect

See also :marks to list all extmarks.

Example: To bind this function to the vim-scriptease inspired zS in Normal mode:

```
vim.keymap.set('n', 'zS', vim.show_pos)
```

Attributes:
Since: 0.9.0

Parameters:

- {buf} (integer?) defaults to the current buffer

- {row} (integer?) row to inspect, 0-based. Defaults to the row of the current cursor
- {col} (integer?) col to inspect, 0-based. Defaults to the col of the current cursor
- {filter} (table?) A table with the following fields:
  - {extmarks} (boolean|"all", default: true) Include extmarks. When all, then extmarks without a hl_group will also be included.
  - {semantic_tokens} (boolean, default: true) Include semantic token highlights.
  - {syntax} (boolean, default: true) Include syntax based highlight groups.
  - {treesitter} (boolean, default: true) Include treesitter based highlight groups.

Lua module: vim.base64 vim.base64

vim.base64.decode({str}) vim.base64.decode()
Decode a Base64 encoded string.

Parameters:
- {str} (string) Base64 encoded string

Return:
(string) Decoded string

vim.base64.encode({str}) vim.base64.encode()
Encode {str} using Base64.

Parameters:
- {str} (string) String to encode

Return:
(string) Encoded string

Lua module: vim.filetype vim.filetype

vim.filetype.add({filetypes}) vim.filetype.add()
Add new filetype mappings.

Filetype mappings can be added either by extension or by filename (either the "tail" or the full file path). The full file path is checked first, followed by the file name. If a match is not found using the filename, then the filename is matched against the list of lua-patterns (sorted by priority) until a match is found. Lastly, if pattern matching does not find a filetype, then the file extension is used. Extension mappings match only the text after the final dot in the filename.

The filetype can be either a string (in which case it is used as the filetype directly) or a function. If a function, it takes the full path and buffer number of the file as arguments (along with captures from the matched pattern, if any) and should return a string that will be used as the buffer's filetype. Optionally, the function can return a second function value which, when called, modifies the state of the buffer. This can be used to, for example, set filetype-specific buffer variables. This function will be called by Nvim before setting the buffer's filetype.

Filename patterns can specify an optional priority to resolve cases when a file path matches multiple patterns. Higher priorities are matched first. When omitted, the priority defaults to 0. A pattern can contain environment variables of the form "${SOME_VAR}" that will be automatically expanded. If the environment variable is not set, the pattern won't be matched.

See $VIMRUNTIME/lua/vim/filetype.lua for more examples.

Example:

```
vim.filetype.add({
  extension = {
    foo = 'fooscript',
    bar = function(path, bufnr)
      if some_condition() then
        return 'barscript', function(bufnr)
          -- Set a buffer variable
          vim.b[bufnr].barscript_version = 2
        end
      end
      return 'bar'
    end,
  },
  filename = {
    ['.foorc'] = 'toml',
    ['/etc/foo/config'] = 'toml',
  },
  pattern = {
    ['.*/etc/foo/.*'] = 'fooscript',
    -- Using an optional priority
    ['.*/etc/foo/.*%.conf'] = { 'dosini', { priority = 10 } },
    -- A pattern containing an environment variable
    ['${XDG_CONFIG_HOME}/foo/git'] = 'git',
    ['.*README.(%a+)'] = function(path, bufnr, ext)
      if ext == 'md' then
        return 'markdown'
      elseif ext == 'rst' then
        return 'rst'
      end
    end,
  },
})
```

To add a fallback match on contents, use

```
vim.filetype.add {
  pattern = {
    ['.*'] = {
      function(path, bufnr)
        local content = vim.api.nvim_buf_get_lines(bufnr, 0, 1, false)[1] or ''
        if vim.regex([[^#!.*\\<mine\\>]]):match_str(content) ~= nil then
          return 'mine'
        elseif vim.regex([[\\<drawing\\>]]):match_str(content) ~= nil then
          return 'drawing'
        end
      end,
      { priority = -math.huge },
    },
  },
}
```

Parameters:
- {filetypes} (table) A table containing new filetype maps (see example).
  - {extension}? (vim.filetype.mapping)
  - {filename}? (vim.filetype.mapping)
  - {pattern}? (vim.filetype.mapping)

vim.filetype.get_option() vim.filetype.get_option({filetype}, {option}) Get the default option value for a {filetype}.

The returned value is what would be set in a new buffer after 'filetype' is set, meaning it should respect all FileType autocmds and ftplugin files.

Example:

```
vim.filetype.get_option('vim', 'commentstring')
```

Note: this uses nvim_get_option_value() but caches the result. This means ftplugin and FileType autocommands are only triggered once and may not reflect later changes.

Attributes:
Since: 0.9.0

Parameters:
- {filetype} (string) Filetype
- {option} (string) Option name

Return:
(string|boolean|integer) Option value

vim.filetype.inspect() vim.filetype.inspect()
Inspect the current state of the filetype registry.

Returns a copy of the internal tables used for filetype detection by extension, filename, or pattern. **Note:** Due to the dynamic nature of filetype detection, this is only useful for checking whether a certain extension, filename, or pattern has been registered so far. In addition, the pattern table is in an internal format optimized for fast lookup. Prefer vim.filetype.match() for checking the detected filetype for a given pattern.

Return:
({ extension: vim.filetype.mapping, filename: vim.filetype.mapping, pattern: table<string, vim.filetype.mapping> })

vim.filetype.match({args}) vim.filetype.match()
Perform filetype detection.

The filetype can be detected using one of three methods:
1. Using an existing buffer
2. Using only a file name
3. Using only file contents

Of these, option 1 provides the most accurate result as it uses both the buffer's filename and (optionally) the buffer contents. Options 2 and 3 can be used without an existing buffer, but may not always provide a match in cases where the filename (or contents) cannot unambiguously determine the filetype.

Each of the three options is specified using a key to the single argument of this function. Example:

```
-- Using a buffer number
vim.filetype.match({ buf = 42 })

-- Override the filename of the given buffer
vim.filetype.match({ buf = 42, filename = 'foo.c' })

-- Using a filename without a buffer
vim.filetype.match({ filename = 'main.lua' })

-- Using file contents
vim.filetype.match({ contents = {'#!/usr/bin/env bash'} })
```

Parameters:
- {args} (table) Table specifying which matching strategy to use. Accepted keys are:
  - {buf}? (integer) Buffer number to use for matching. Mutually exclusive with {contents}
  - {contents}? (string[]) An array of lines representing file contents to use for matching. Can be used with {filename}. Mutually exclusive with {buf}.
  - {filename}? (string) Filename to use for matching. When {buf} is given, defaults to the filename of the given buffer number. The file need not actually exist in the filesystem. When used without {buf} only the name of the file is used for filetype matching. This may

result in failure to detect the filetype in cases where the filename alone is not enough to disambiguate the filetype.

Return (multiple):
(`string?`) The matched filetype, if any. (`function?`) A function `fun(buf: integer)` that modifies buffer state when called (for example, to set some filetype specific buffer variables). (`boolean?`) true if a match was found by falling back to a generic filetype (i.e., ".conf"), which indicates the filetype should be set with `:setf FALLBACK conf`. See :setfiletype.

Lua module: vim.fs

vim.fs

vim.fs.copy()

Use filecopy() or uv.fs_copyfile() to performantly copy an existing file.

Example:

```
vim.fn.filecopy('foo.txt', 'bar.txt')
```

vim.fs.exists()

Use uv.fs_stat() to check a file's type, and whether it exists.

Example:

```
if vim.uv.fs_stat(file) then
  vim.print('file exists')
end
```

vim.fs.read()

You can use readblob() to get a file's contents without explicitly opening/closing it. Or use io.lines() to iterate lines in a text file.

Example:

```
vim.print(vim.fn.readblob('.git/config'))
```

vim.fs.write()

You can use writefile() to write a file without explicitly opening/closing it.

Example:

```
vim.fn.writefile('foo\0bar', 'data.bin', 'b')
```

vim.fs.abspath({path}, {opts}) vim.fs.abspath()
Converts path to an absolute path. Expands tilde (~) at the beginning of the path (unless plain=true). Does not check if the path exists, normalize the path, resolve symlinks or hardlinks (including "." and ".."), or expand environment variables. If the path is already absolute, it is returned unchanged. Converts \ path separators to /.

Attributes:
Since: 0.11.0

Parameters:
- {path} (`string`) Path
- {opts} (`table?`) A table with the following fields:
  - {cwd}? (`string`) Resolve the path relative to this directory.
  - {plain}? (`boolean`) Do not expand tilde (~).

Return:
(`string`) Absolute path

vim.fs.basename({file}) vim.fs.basename()
Gets the basename of the given path (not expanded/resolved).

Attributes:
Since: 0.8.0

Parameters:
- {file} (`string?`) Path

Return:
(`string?`) Basename of `file`

vim.fs.dir({path}, {opts}) vim.fs.dir()
Gets an iterator over items found in path (normalized via vim.fs.normalize()).

Example:

```
for name, type, err in vim.fs.dir(path, { err = true }) do
  if err then
    -- Failed to scan directory {name} (may be the root {path} itself).
  end
end
```

Attributes:
Since: 0.8.0

Parameters:
- {path} (`string`) Directory to iterate over, expanded (unless `opts.plain=true`) and normalized.
- {opts} (`table?`) A table with the following fields:
  - {depth}? (`integer`, default: 1) How deep to traverse.

- {err}? (boolean, default: false) Report errors via the iterator's third value ("err"), instead of silently skipping.
- {follow}? (boolean, default: false) Follow symbolic links.
- {plain}? (boolean, default: false) Do not expand special forms like "~" and "$" in {path}.
- {skip}? (fun(dir_name: string): boolean) Predicate to control traversal. Return false to stop searching the current directory. Only useful when depth > 1 Return an iterator over the items located in {path}

Return:
(fun(): string?, string?, string?) Iterator over items in {path}, yielding (name, type, err):
- name: Basename of the item relative to {path}.
- type: One of: "file", "directory", "link", "fifo", "socket", "char", "block", "unknown".
- err: Error string, or nil. Only if opts.err=true. If the root {path} itself could not be scanned, yields a single (name, nil, err) item.

vim.fs.dirname({file}) vim.fs.dirname()
Gets the parent directory of the given path (not expanded/resolved, the caller must do that).

Attributes:
Since: 0.8.0

Parameters:
- {file} (string?) Path

Return:
(string?) Parent directory of file

vim.fs.ext({file}, {opts}) vim.fs.ext()
Return the file's last extension, if any.

Similar to fnamemodify() with the ::e modifier. The extension does not include a leading period.

Examples:

```
vim.fs.ext('archive.tar.gz') -- 'gz'
vim.fs.ext('~/.git') -- ''
vim.fs.ext('plugin/myplug.lua') -- 'lua'
```

Attributes:
Since: 0.12.0

Parameters:
- {file} (string) Path
- {opts} (table?) Reserved for future use

Return:
(string) Extension of {file}

vim.fs.find({names}, {opts}) vim.fs.find()
Find files or directories (or other items as specified by opts.type) in the given path.

Finds items given in {names} starting from {path}. If {upward} is "true" then the search traverses upward through parent directories; otherwise, the search traverses downward. Note that downward searches are recursive and may search through many directories! If {stop} is non-nil, then the search stops when the directory given in {stop} is reached. The search terminates when {limit} (default 1) matches are found. You can set {type} to "file", "directory", "link", "socket", "char", "block", or "fifo" to narrow the search to find only that type.

Examples:

```
-- List all test directories under the runtime directory.
local dirs = vim.fs.find(
  { 'test', 'tst', 'testdir' },
  { limit = math.huge, type = 'directory', path = './runtime/' }
)

-- Get all "lib/*.cpp" and "lib/*.hpp" files, using Lua patterns.
-- Or use `vim.glob.to_lpeg(…):match(…)` for glob/wildcard matching.
local files = vim.fs.find(function(name, path)
  return name:match('.*%.[ch]pp$') and path:match('[/\\]lib$')
end, { limit = math.huge, type = 'file' })
```

Attributes:
Since: 0.8.0

Parameters:
- {names} (string|string[]|fun(name: string, path: string): boolean) Names of the items to find. Must be base names, paths and globs are not supported when {names} is a string or a table. If {names} is a function, it is called for each traversed item with args:
  - name: base name of the current item
  - path: full path of the directory containing the current item

The function should return true if the given item is considered a match.
- {opts} (table?) Optional keyword arguments:
  - {follow}? (boolean, default: false) Follow symbolic links.
  - {limit}? (number, default: 1) Stop searching after this many matches. Use math.huge for "unlimited".
  - {path}? (string) Path to begin searching from, defaults to current-directory. Not expanded.
  - {stop}? (string) Stop searching when this directory is reached. The directory itself is not searched.
  - {type}? (string) Find only items of the given type. If omitted, all items that match names are included. With follow, a symlink has the type of its target.
  - {upward}? (boolean, default: false) Search upward through parent directories. Otherwise, search child directories (recursively).

Return (multiple):
(string[]) Normalized paths vim.fs.normalize() of all matching items. (string[]) Errors collected while searching.

vim.fs.joinpath({...}) vim.fs.joinpath()

Concatenates partial paths (one absolute or relative path followed by zero or more relative paths). Slashes are normalized: redundant slashes are removed, and (on Windows) backslashes are replaced with forward-slashes. Empty segments are removed. Paths are not expanded/resolved.

Examples:

- "foo/", "/bar" => "foo/bar"
- "", "after/plugin" => "after/plugin"
- Windows: "a\foo\", "\bar" => "a/foo/bar"

Attributes:
Since: 0.10.0

Parameters:
- `{...}` (`string`)

Return:
(`string`)

vim.fs.mkdir({path}, {opts}) vim.fs.mkdir()

Creates a directory.

Attributes:
Since: 0.13.0

Parameters:
- `{path}` (`string`) Path to create (not expanded/resolved).
- `{opts}` (`table?`) Optional keyword arguments.
  - `{mode}?` (`integer`, default: 493) Permission bits for newly-created directories.
  - `{parents}?` (`boolean`, default: `false`) Create intermediate directories as necessary.

vim.fs.normalize({path}, {opts}) vim.fs.normalize()

Normalize a path to a standard format. Expands environment variables, and tilde "~" at the beginning of the path. Resolves "." and ".." components, except when the path is relative and resolving it would produce an absolute path.

- "." as the only part in a relative path:
  - "." => "."
  - "./." => "."
- ".." when it leads outside the current directory
  - "foo/../../bar" => "../bar"
  - "../../foo" => "../../foo"
- ".." in the root directory returns the root directory.
  - "/../../" => "/"

On Windows, backslashes (`\`) are converted to forward slashes (`/`).

Examples:

```
[[C:\Users\jdoe]]                         --> "C:/Users/jdoe"
"~/src/neovim"                            --> "/home/jdoe/src/neovim"
"$XDG_CONFIG_HOME/nvim/init.vim"          --> "/Users/jdoe/.config/nvim/init.vim"
"~/src/nvim/api/../tui/./tui.c"           --> "/home/jdoe/src/nvim/tui/tui.c"
"./foo/bar"                               --> "foo/bar"
"foo/../../../bar"                        --> "../../bar"
"/home/jdoe/../../../bar"                 --> "/bar"
"C:foo/../../baz"                         --> "C:../baz"
"C:/foo/../../baz"                        --> "C:/baz"
[[\\?\UNC\server\share\foo\..\..\..\bar]] --> "//?/UNC/server/share/bar"
```

Attributes:
Since: 0.8.0

Parameters:
- `{path}` (`string`) Path to normalize
- `{opts}` (`table?`) A table with the following fields:
  - `{plain}?` (`boolean`, default: `false`) Do not expand environment variables and leading tildes "~".
  - `{win}?` (`boolean`, default: `true` in Windows, `false` otherwise) Path is a Windows path.

Return:
(`string`) Normalized path

vim.fs.parents({start}) vim.fs.parents()

Iterate over all the parents of the given path (not expanded/resolved, the caller must do that).

Example:

```
local root_dir
for dir in vim.fs.parents(vim.api.nvim_buf_get_name(0)) do
  if vim.fn.isdirectory(dir .. '/.git') == 1 then
    root_dir = dir
    break
  end
end

if root_dir then
  print('Found git repository at', root_dir)
end
```

Attributes:

Since: 0.8.0

Parameters:
- {start} (string) Initial path.

Return (multiple):
(fun(_, dir: string): string?) Iterator (nil) (string?)

vim.fs.relpath({base}, {target}, {opts}) vim.fs.relpath()
Gets target path relative to base, or nil if base is not an ancestor.

Example:

```
vim.fs.relpath('/var', '/var/lib') -- 'lib'
vim.fs.relpath('/var', '/usr/bin') -- nil
```

Attributes:
Since: 0.11.0

Parameters:
- {base} (string)
- {target} (string)
- {opts} (table?) Reserved for future use

Return:
(string?)

vim.fs.rm({path}, {opts}) vim.fs.rm()
Removes a file or directory.

Removes symlinks without touching the origin. To remove the origin, resolve it explicitly with uv.fs_realpath():

```
vim.fs.rm(vim.uv.fs_realpath('symlink-dir'), { recursive = true })
```

Attributes:
Since: 0.11.0

Parameters:
- {path} (string) Path to remove (not expanded/resolved).
- {opts} (table?) A table with the following fields:
  - {force}? (boolean) Ignore nonexistent files and arguments.
  - {recursive}? (boolean) Remove directory contents recursively.

vim.fs.root({source}, {marker}) vim.fs.root()
Find the first parent directory containing a specific "marker", relative to a file path or buffer.

If the buffer is unnamed (has no backing file) or has a non-empty 'buftype' then the search begins from Nvim's current-directory.

Examples:

```
-- Find the root of a Python project, starting from file 'main.py'
vim.fs.root(vim.fs.joinpath(vim.env.PWD, 'main.py'), {'pyproject.toml', 'setup.py' })

-- Find the root of a git repository
vim.fs.root(0, '.git')

-- Find the parent directory containing any file with a .csproj extension
vim.fs.root(0, function(name, path)
  return vim.fs.ext(name) == 'csproj'
end)

-- Find the first ancestor directory containing EITHER "stylua.toml" or ".luarc.json"; if
-- not found, find the first ancestor containing ".git":
vim.fs.root(0, { { 'stylua.toml', '.luarc.json' }, '.git' })
```

Attributes:
Since: 0.10.0

Parameters:
- {source} (integer|string) Buffer number (0 for current buffer) or file path (absolute or relative, expanded via abspath()) to begin the search from.
- {marker} ((string|string[]|fun(name: string, path: string): boolean)[]|string|fun(name: string, path: string): boolean) Filename, function, or list thereof, that decides how to find the root. To indicate "equal priority", specify items in a nested list { { 'a.txt', 'b.lua' }, … }. A function item must return true if name and path are a match. Each item (which may itself be a nested list) is evaluated in-order against all ancestors, until a match is found.

Return:
(string?) Directory path containing one of the given markers, or nil if no directory was found.

vim.fs.slug({path}, {opts}) vim.fs.slug()
Gets a filesystem-safe, mnemonic slug (readable prefix + short hash) of an arbitrary filepath or other "identity string".
- The input is normalized so equivalent paths produce the same result.
- A hash of the normalized input is appended to prevent collisions.
- Unsafe chars are replaced with "-".
- $HOME is replaced with "~".
- URIs (scheme://...) are prefixed with "=uri-{scheme}-".
- UNC paths (Windows) are prefixed with "=unc-".
- If opts.maxlen is exceeded, the result will be truncated to {head}~~~{tail}-{hash8}.
- If the sanitized name is empty, the reserved label =special will be used.

Examples:

```
vim.print(vim.fs.slug('/tmp/test/foo.md'))           --> "tmp-test-foo.md-{hash}"
vim.print(vim.fs.slug('C:/src/project/main.c'))      --> "C--src-project-main.c-{hash}"
vim.print(vim.fs.slug(vim.fn.expand('~/file.txt')))  --> "~-file.txt-{hash}"
vim.print(vim.fs.slug('---'))                        --> "=special-{hash}"
vim.print(vim.fs.slug('term://foo/bar//123:bash'))   --> "=uri-term-foo-bar-123-bash-{hash}"
vim.print(vim.fs.slug(('/a/very/long/path'):rep(10) .. '/file.txt', { maxlen = 60 }))
   --> "a-very-long-~~~-path-a-very-long-path-file.txt-{hash}"
```

Attributes:
Since: 0.13.0

Parameters:
- {path} (string) Filepath (or other identity string).
- {opts} ({ maxlen?: integer }?)
  - maxlen: (integer, default: 180) Max length (bytes) of the result.

Return:
(string) Filesystem-safe, mnemonic slug.

Lua module: vim.glob

vim.glob

Glob-to-LPeg Converter (Peglob) This module converts glob patterns to LPeg patterns according to the LSP 3.17 specification: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#pattern

Glob grammar overview:
- * to match zero or more characters in a path segment
- ? to match on one character in a path segment
- ** to match any number of path segments, including none
- {} to group conditions (e.g. *.{ts,js} matches TypeScript and JavaScript files)
- [] to declare a range of characters to match in a path segment (e.g., example.[0-9] to match on example.0, example.1, …)
- [!...] to negate a range of characters to match in a path segment (e.g., example.[!0-9] to match on example.a, example.b, but not example.0)

Additional constraints:
- A Glob pattern must match an entire path, with partial matches considered failures.
- The pattern only determines success or failure, without specifying which parts correspond to which characters.
- A path segment is the portion of a path between two adjacent path separators (/), or between the start/end of the path and the nearest separator.
- The ** (globstar) pattern matches zero or more path segments, including intervening separators (/). Within pattern strings, ** must be delimited by path separators (/) or pattern boundaries and cannot be adjacent to any characters other than /. If ** is not the final element, it must be followed by /.
- {} (braced conditions) contains valid Glob patterns as branches, separated by commas. Commas are exclusively used for separating branches and cannot appear within a branch for any other purpose. Nested {} structures are allowed, but {} must contain at least two branches—zero or one branch is not permitted.
- In [] or [!...], a character range consists of character intervals (e.g., a-z) or individual characters (e.g., w). A range including / won't match that character.

vim.glob.to_lpeg({pattern}) vim.glob.to_lpeg()
Parses a raw glob into an lua-lpeg pattern.

Parameters:
- {pattern} (string) The raw glob pattern

Return:
(vim.lpeg.Pattern) An lua-lpeg representation of the pattern

Lua module: vim.hl

vim.hl

vim.hl.hl_op({opts}) vim.hl.hl_op()
Highlights the affected text region during a TextYankPost or TextPutPost event.

Add this to your init.vim:

```
autocmd TextYankPost * silent! lua vim.hl.hl_op {higroup='Visual', timeout=300}
autocmd TextPutPost  * silent! lua vim.hl.hl_op {higroup='Visual', timeout=300}
```

Parameters:
- {opts} (table?) Optional parameters
  - event (default: vim.v.event) Event structure.
  - higroup (default: "IncSearch") Highlight group for the text region.
  - on_macro (default: false) Highlight during macro execution.
  - on_visual (default: true) Highlight during Visual mode.
  - priority (default: vim.hl.priorities.user) Integer priority.
  - timeout (default: 150) Time in ms before highlight is cleared.

vim.hl.priorities vim.hl.priorities
Table with default priorities used for highlighting:
- syntax: 50, used for standard syntax highlighting
- treesitter: 100, used for treesitter-based highlighting
- semantic_tokens: 125, used for LSP semantic token highlighting
- diagnostics: 150, used for code analysis such as diagnostics

- `user`: 200, used for user-triggered highlights such as LSP document symbols or `hl_op` autocommands

vim.hl.range()

vim.hl.range({buf}, {ns}, {hlgroup}, {start}, {finish}, {opts}) Apply highlight group to range of text.

Parameters:
- {buf} (integer) Buffer number to apply highlighting to
- {ns} (integer) Namespace to add highlight to
- {hlgroup} (integer|integer[]|string|string[]) Highlight group used for the text range. See the `hl_group` option in nvim_buf_set_extmark().
- {start} ([integer,integer]|string) Start of region as a (line, column) tuple or string accepted by getpos()
- {finish} ([integer,integer]|string) End of region as a (line, column) tuple or string accepted by getpos()
- {opts} (table?) A table with the following fields:
  - {inclusive}? (boolean, default: false) Indicates whether the range is end-inclusive
  - {priority}? (integer, default: vim.hl.priorities.user) Highlight priority
  - {regtype}? (string, default: 'v' i.e. charwise) Type of range. See getregtype()
  - {timeout}? (integer, default: -1 no timeout) Time in ms before highlight is cleared

Return (multiple):
(uv.uv_timer_t?) range_timer A timer which manages how much time the highlight has left (fun()?) range_clear A function which allows clearing the highlight manually. nil is returned if timeout is not specified

Lua module: vim.iter

vim.iter

vim.iter() is an interface for iterables: it wraps a table or function argument into an Iter object with methods (such as Iter:filter() and Iter:map()) that transform the underlying source data. These methods can be chained to create iterator "pipelines": the output of each pipeline stage is input to the next stage. The first stage depends on the type passed to `vim.iter()`:
- Lists or arrays (lua-list) yield only the value of each element.
  - Holes (nil values) are allowed (but discarded).
  - Use pairs() to treat array/list tables as dicts (preserve holes and non-contiguous integer keys): `vim.iter(pairs(…))`.
  - Use Iter:enumerate() to also pass the index to the next stage.
    - Or initialize with ipairs(): `vim.iter(ipairs(…))`.
- Non-list tables (lua-dict) yield both the key and value of each element.
- Function iterators yield all values returned by the underlying function.
- Tables with a __call() metamethod are treated as function iterators.

The iterator pipeline terminates when the underlying iterable is exhausted (for function iterators this means it returned nil).

Note: `vim.iter()` scans table input to decide if it is a list or a dict; to avoid this cost you can wrap the table with an iterator e.g. `vim.iter(ipairs({…}))`, but that precludes the use of list-iterator operations such as IterArray:rev()).

Examples:

```
local it = vim.iter({ 1, 2, 3, 4, 5 })
it:map(function(v)
  return v * 3
end)
it:rev()
it:skip(2)
it:totable()
-- { 9, 6, 3 }

-- ipairs() is a function iterator which returns both the index (i) and the value (v)
vim.iter(ipairs({ 1, 2, 3, 4, 5 })):map(function(i, v)
  if i > 2 then return v end
end):totable()
-- { 3, 4, 5 }

local it = vim.iter(vim.gsplit('1,2,3,4,5', ','))
it:map(function(s) return tonumber(s) end)
for i, d in it:enumerate() do
  print(string.format("Column %d is %d", i, d))
end
-- Column 1 is 1
-- Column 2 is 2
-- Column 3 is 3
-- Column 4 is 4
-- Column 5 is 5

vim.iter({ a = 1, b = 2, c = 3, z = 26 }):any(function(k, v)
  return k == 'z'
end)
-- true

local rb = vim.ringbuf(3)
rb:push("a")
rb:push("b")
vim.iter(rb):totable()
-- { "a", "b" }
```

Iter:all({pred}) Iter:all()
Returns true if all items in the iterator match the given predicate.

Attributes:
Since: 0.10.0

Parameters:
- `{pred}` (`fun(v: V1, ...: V...): boolean`) Predicate function. Takes all values returned from the previous stage in the pipeline as arguments and returns true if the predicate matches.

Iter:any(`{pred}`) Iter:any()
Returns true if any of the items in the iterator match the given predicate.

Attributes:
Since: 0.10.0

Parameters:
- `{pred}` (`fun(v: V1, ...: V...): boolean`) Predicate function. Takes all values returned from the previous stage in the pipeline as arguments and returns true if the predicate matches.

Iter:count() Iter:count()
Drains the iterator, counting the number of iterations and returning it.

Attributes:
Since: 0.13.0

Return:
(`integer`)

Iter:each(`{f}`) Iter:each()
Calls a function once for each item in the pipeline, draining the iterator.

For functions with side effects. To modify the values in the iterator, use Iter:map().

Attributes:
Since: 0.10.0

Parameters:
- `{f}` (`fun(v: V1, ...: V...)`) Function to execute for each item in the pipeline. Takes all of the values returned by the previous stage in the pipeline as arguments.

Iter:enumerate() Iter:enumerate()
Yields the item index (count) and value for each item of an iterator pipeline.

For list tables, this is more efficient:

```
vim.iter(ipairs(t))
```

instead of:

```
vim.iter(t):enumerate()
```

Example:

```
local it = vim.iter(vim.gsplit('abc', '')):enumerate()
it:next()
-- 1    'a'
it:next()
-- 2    'b'
it:next()
-- 3    'c'
```

Attributes:
Since: 0.10.0

Overloads:
- `fun<V1, V...>(self: vim.IterArray<V1, V...>): vim.IterArray<integer, V1, V...>`

Return:
(`vim.Iter<integer, V1, V...>`)

Iter:filter(`{f}`) Iter:filter()
Filters an iterator pipeline.

Example:

```
local bufs = vim.iter(vim.api.nvim_list_bufs()):filter(vim.api.nvim_buf_is_loaded)
```

Parameters:
- `{f}` (`fun(v: V1, ...: V...): boolean`) Takes all values returned from the previous stage in the pipeline and returns false or nil if the current iterator element should be removed.

Overloads:
- `fun<V1, V...>(self: vim.IterArray<V1, V...>, f: fun(v: V1, ...: V...): boolean): vim.IterArray<V1, V...>`

Return:
(`vim.Iter<V1, V...>`)

Iter:find(`{f}`) Iter:find()
Find the first value in the iterator that satisfies the given predicate.

Advances the iterator. Returns nil and drains the iterator if no value is found.

Examples:

```
local it = vim.iter({ 3, 6, 9, 12 })
it:find(12)
-- 12

local it = vim.iter({ 3, 6, 9, 12 })
it:find(20)
-- nil

local it = vim.iter({ 3, 6, 9, 12 })
it:find(function(v) return v % 4 == 0 end)
-- 12
```

Attributes:
Since: 0.10.0

Parameters:
- `{f}` (`V1|fun(v: V1, ...: V...): any`)

Return (multiple):
(`V1?`) (`V...`)

Iter:fold(`{init}`, `{f}`) Iter:fold()
Folds ("reduces") an iterator into a single value. Iter:reduce()

Examples:

```
-- Create a new table with only even values
vim.iter({ a = 1, b = 2, c = 3, d = 4 })
  :filter(function(k, v) return v % 2 == 0 end)
  :fold({}, function(acc, k, v)
    acc[k] = v
    return acc
  end) --> { b = 2, d = 4 }

-- Get the "maximum" item of an iterable.
vim.iter({ -99, -4, 3, 42, 0, 0, 7 })
  :fold({}, function(acc, v)
    acc.max = math.max(v, acc.max or v)
    return acc
  end) --> { max = 42 }
```

Attributes:
Since: 0.10.0

Parameters:
- `{init}` (any) Initial value of the accumulator.
- `{f}` (`fun(acc: A, v: V1, ...: V...): A`) Accumulation function.

Return:
(any)

Iter:join(`{delim}`) Iter:join()
Collect the iterator into a delimited string.

Each element in the iterator is joined into a string separated by `{delim}`.

Consumes the iterator.

Attributes:
Since: 0.10.0

Parameters:
- `{delim}` (`string`) Delimiter

Return:
(`string`)

Iter:last() Iter:last()
Drains the iterator and returns the last item.

Example:

```
local it = vim.iter(vim.gsplit('abcdefg', ''))
it:last()
-- 'g'

local it = vim.iter({ 3, 6, 9, 12, 15 })
it:last()
-- 15
```

Attributes:
Since: 0.10.0

Return (multiple):
(`V1?`) (`V...`)

See also:
- IterArray:rpeek()

Iter:map({f}) Iter:map()
Maps the items of an iterator pipeline to the values returned by `f`.

If the map function returns nil, the value is filtered from the iterator.

Example:

```
local it = vim.iter({ 1, 2, 3, 4 }):map(function(v)
  if v % 2 == 0 then
    return v * 3
  end
end)
it:totable()
-- { 6, 12 }
```

Attributes:
Since: 0.10.0

Parameters:
- `{f}` (`fun(v: V1, ...: V...): K2, V2...`) Mapping function. Takes all values returned from the previous stage in the pipeline as arguments and returns one or more new values, which are used in the next pipeline stage. Nil return values are filtered from the output.

Overloads:
- `fun<V1, V..., K1, K...>(self: vim.IterArray<V1, V...>, f: fun(v: V1, ...: V...): K1, K...): vim.IterArray<K1, K...>`

Return:
(`vim.Iter<K2, V2...>`)

Iter:next() Iter:next()
Gets the next value from the iterator.

Example:

```
local it = vim.iter(string.gmatch('1 2 3', '%d+')):map(tonumber)
it:next()
-- 1
it:next()
-- 2
it:next()
-- 3
```

Attributes:
Since: 0.10.0

Return (multiple):
(`V1?`) (`V...`)

Iter:nth({n}) Iter:nth()
Gets the nth value of an iterator (and advances to it).

If n is negative, offsets from the end of a list-iterator.

Example:

```
local it = vim.iter({ 3, 6, 9, 12 })
it:nth(2)
-- 6
it:nth(2)
-- 12

local it2 = vim.iter({ 3, 6, 9, 12 })
it2:nth(-2)
-- 9
it2:nth(-2)
-- 3
```

Attributes:
Since: 0.10.0

Parameters:
- `{n}` (`integer`) Index of the value to return. May be negative if the source is a list-iterator.

Return (multiple):
(`V1?`) (`V...`)

Iter:peek() Iter:peek()
Gets the next value from the iterator without consuming it.

The value returned by Iter:peek() will be returned again by the next call to Iter:next().

Example:

```
local it = vim.iter({ 3, 6, 9, 12 })
it:peek()
-- 3
it:peek()
-- 3
it:next()
-- 3
```

Attributes:
Since: 0.10.0

Return (multiple):
(V1?) (`V...`)

Iter:skip({n}) Iter:skip()
Skips n values of an iterator pipeline, or skips values while a predicate returns lua-truthy.

When a predicate is used, skipping stops at the first value for which the predicate returns non-truthy. That value is not consumed and will be returned by the next call to Iter:next()

Example:

```
local it = vim.iter({ 3, 6, 9, 12 }):skip(2)
it:next()
-- 9

local function pred(x) return x < 10 end
local it2 = vim.iter({ 3, 6, 9, 12 }):skip(pred)
it2:next()
-- 12
```

Attributes:
Since: 0.10.0

Parameters:
- `{n}` (`integer|fun(v: V1, ...: V...): boolean`) Number of values to skip or a predicate.

Overloads:
- `fun<V1, V...>(self: vim.IterArray<V1, V...>, n: integer|fun(v: V1, ...: V...): boolean): vim.IterArray<V1, V...>`

Return:
(`vim.Iter<V1, V...>`)

Iter:take({n}) Iter:take()
Transforms an iterator to yield only the first n values, or all values satisfying a predicate.

Example:

```
local it = vim.iter({ 1, 2, 3, 4 }):take(2)
it:next()
-- 1
it:next()
-- 2
it:next()
-- nil

local function pred(x) return x < 2 end
local it2 = vim.iter({ 1, 2, 3, 4 }):take(pred)
it2:next()
-- 1
it2:next()
-- nil
```

Attributes:
Since: 0.10.0

Parameters:
- `{n}` (`integer|fun(v: V1, ...: V...): boolean`) Number of values to take or a predicate.

Overloads:
- `fun<V1, V...>(self: vim.IterArray<V1, V...>, n: integer|fun(v: V1, ...: V...): boolean): vim.IterArray<V1, V...>`

Return:
(`vim.Iter<V1, V...>`)

Iter:totable() Iter:totable()
Collect the iterator into a table.

The resulting table depends on the initial source in the iterator pipeline. Array-like tables and function iterators will be collected into an array-like table. If multiple values are returned from the final stage in the iterator pipeline, each value will be included in a table.

Examples:

```
vim.iter(string.gmatch('100 20 50', '%d+')):map(tonumber):totable()
-- { 100, 20, 50 }

vim.iter({ 1, 2, 3 }):map(function(v) return v, 2 * v end):totable()
-- { { 1, 2 }, { 2, 4 }, { 3, 6 } }

vim.iter({ a = 1, b = 2, c = 3 }):filter(function(k, v) return v % 2 ~= 0 end):totable()
-- { { 'a', 1 }, { 'c', 3 } }
```

The generated table is an array-like table with consecutive, numeric indices. To create a map-like table with arbitrary keys, use Iter:fold().

Attributes:
Since: 0.10.0

Return:
(`any[]`)

Iter:unique({key}) Iter:unique()
Removes duplicate values from an iterator pipeline.

Only the first occurrence of each value is kept.

Accepts an optional key argument, which if provided is called for each value in the iterator to compute a hash key for uniqueness comparison. This is useful for deduplicating table values or complex objects. If key returns `nil` for a value, that value will be considered unique, even if multiple values return `nil`.

If a function-based iterator returns multiple arguments, uniqueness is checked based on the first return value. To change this behavior, specify key.

Examples:

```
vim.iter({ 1, 2, 2, 3, 2 }):unique():totable()
-- { 1, 2, 3 }

vim.iter({ {id=1}, {id=2}, {id=1} })
  :unique(function(x)
    return x.id
  end)
  :totable()
-- { {id=1}, {id=2} }
```

Attributes:
Since: 0.12.0

Parameters:
- {key} (`fun(v: V1, ...: V...): any?`) Optional hash function to determine uniqueness of values.

Overloads:
- `fun<V1, V...>(self: vim.IterArray<V1, V...>, key: fun(v: V1, ...: V...): any?): vim.IterArray<V1, V...>`

Return:
(`vim.Iter<V1, V...>`)

See also:
- vim.list.unique()

IterArray:flatten({depth}) IterArray:flatten()
Flattens a list-iterator, un-nesting nested values up to the given {depth}. Errors if it attempts to flatten a dict-like value.

Examples:

```
vim.iter({ 1, { 2 }, { { 3 } } }):flatten():totable()
-- { 1, 2, { 3 } }

vim.iter({1, { { a = 2 } }, { 3 } }):flatten():totable()
-- { 1, { a = 2 }, 3 }

vim.iter({ 1, { { a = 2 } }, { 3 } }):flatten(math.huge):totable()
-- error: attempt to flatten a dict-like table
```

Attributes:
Since: 0.10.0

Parameters:
- {depth} (`integer?`) Depth to which list-iterator should be flattened (defaults to 1)

Return:
(`vim.IterArray<V1, V...>`)

IterArray:pop() IterArray:pop()
"Pops" a value from a list-iterator (gets the last value and decrements the tail).

Example:

```
local it = vim.iter({1, 2, 3, 4})
it:pop()
-- 4
it:pop()
-- 3
```

Attributes:
Since: 0.10.0

Return (multiple):
(V1?) (`V...`)

IterArray:rev() IterArray:rev()
Reverses a list-iterator pipeline.

Example:

```
local it = vim.iter({ 3, 6, 9, 12 }):rev()
it:totable()
-- { 12, 9, 6, 3 }
```

Attributes:
Since: 0.10.0

Return:
(`vim.IterArray<V1, V...>`)

IterArray:rfind({f}) IterArray:rfind()
Gets the first value satisfying a predicate, from the end of a list-iterator.

Advances the iterator. Returns nil and drains the iterator if no value is found.

Examples:

```
local it = vim.iter({ 1, 2, 3, 2, 1 }):enumerate()
it:rfind(1)
-- 5    1
it:rfind(1)
-- 1    1
```

Attributes:
Since: 0.10.0

Parameters:
- `{f}` (`V1|fun(v: V1, ...: V...): boolean`)

Return (multiple):
(`V1?`) (`V...`)

See also:
- Iter:find()

IterArray:rpeek() IterArray:rpeek()
Gets the last value of a list-iterator without consuming it.

Example:

```
local it = vim.iter({1, 2, 3, 4})
it:rpeek()
-- 4
it:rpeek()
-- 4
it:pop()
-- 4
```

Attributes:
Since: 0.10.0

Return (multiple):
(`V1?`) (`V...`)

See also:
- Iter:last()

IterArray:rskip({n}) IterArray:rskip()
Discards n values from the end of a list-iterator pipeline.

Example:

```
local it = vim.iter({ 1, 2, 3, 4, 5 }):rskip(2)
it:next()
-- 1
it:pop()
-- 3
```

Attributes:
Since: 0.10.0

Parameters:
- `{n}` (`integer`)

Return:
(`vim.IterArray<V1, V...>`)

IterArray:slice({`first`}, {`last`}) IterArray:slice()
Sets the start and end of a list-iterator pipeline.

Equivalent to `:skip(first - 1):rskip(len - last + 1)`.

Attributes:
Since: 0.10.0

Parameters:
- `{first}` (`integer`)
- `{last}` (`integer`)

Return:
(`vim.IterArray<V1, V...>`)

Lua module: vim.json vim.json

This module provides encoding and decoding of Lua objects to and from JSON-encoded strings. Supports vim.NIL and vim.empty_dict().

vim.json.decode({str}, {opts}) vim.json.decode()
Decodes (or "unpacks") stringified JSON to a Lua object.
- Decodes JSON "null" as vim.NIL (controllable by {opts}, see below).
- Decodes empty object as vim.empty_dict().
- Decodes empty array as {} (empty Lua table).

Example:

```
vim.print(vim.json.decode('{"bar":[],"foo":{},"zub":null}'))
-- { bar = {}, foo = vim.empty_dict(), zub = vim.NIL }
```

Parameters:
- {str} (string) Stringified JSON data.
- {opts} (table?) A table with the following fields:
  - {luanil}? ({ object?: boolean, array?: boolean }, default: nil) Convert JSON "null" in objects/arrays to Lua nil instead of vim.NIL.
  - {skip_comments}? (boolean, default: false) Allows JavaScript-style comments within JSON data. Comments are treated as whitespace and may appear anywhere whitespace is valid in JSON. Supports single-line comments beginning with '//' and block comments enclosed with '/' and '/'.

Return:
(any)

vim.json.encode({obj}, {opts}) vim.json.encode()
Encodes (or "packs") a Lua object to stringified JSON.

Example: Implement a basic 'formatexpr' for JSON, so gq with a motion formats JSON in a buffer. (The motion must operate on a valid JSON object.)

```
function _G.fmt_json()
  local indent = vim.bo.expandtab and (' '):rep(vim.o.shiftwidth) or '\t'
  local lines = vim.api.nvim_buf_get_lines(0, vim.v.lnum - 1, vim.v.lnum + vim.v.count - 1, true)
  local o = vim.json.decode(table.concat(lines, '\n'))
  local stringified = vim.json.encode(o, { indent = indent, sort_keys = true })
  lines = vim.split(stringified, '\n')
  vim.api.nvim_buf_set_lines(0, vim.v.lnum - 1, vim.v.count, true, lines)
end
vim.o.formatexpr = 'v:lua.fmt_json()'
```

Parameters:
- {obj} (any)
- {opts} (table?) A table with the following fields:
  - {escape_slash}? (boolean, default: false) Escape slash characters "/" in string values.
  - {indent}? (string, default: "") If non-empty, each value in a JSON object or array starts on a new line and is prefixed with indent per nesting level. Adds a space after colons. Does not add a final newline.
  - {sort_keys}? (boolean, default: false) Sort object keys in alphabetical order.

Return:
(string)

Lua module: vim.keymap vim.keymap

vim.keymap.del({modes}, {lhs}, {opts}) vim.keymap.del()
Removes a mapping, or removes each (mode, lhs) pair if lhs is a list.

Examples:

```
vim.keymap.del('n', 'lhs')
vim.keymap.del({'n', 'i', 'v'}, '<leader>w', { buf = 5 })
```

Parameters:
- {modes} (string|string[])
- {lhs} (string|string[])
- {opts} (table?) A table with the following fields:
  - {buf}? (integer) Remove a mapping from the given buffer. 0 for current.
  - {lhs}? (boolean) When true, only match {lhs}, not {rhs}.

See also:
- vim.keymap.set()

vim.keymap.set({modes}, {lhs}, {rhs}, {opts}) vim.keymap.set()
Defines a mapping of keycodes to a function or keycodes. If lhs is a list, defines a mapping for each (mode, lhs) pair.

Examples:

```
-- Map "x" to a Lua function:
vim.keymap.set('n', 'x', function() print('real lua function') end)
-- Map "<leader>x" to multiple modes for the current buffer:
vim.keymap.set({'n', 'v'}, '<leader>x', vim.lsp.buf.references, { buf = 0 })
-- Map <Tab> to an expression (|:map-<expr>|):
vim.keymap.set('i', '<Tab>', function()
  return vim.fn.pumvisible() == 1 and '<C-n>' or '<Tab>'
end, { expr = true })
-- Map "[%%" to a <Plug> mapping:
vim.keymap.set('n', '[%%', '<Plug>(MatchitNormalMultiBackward)')

-- Use `getregionpos(getpos('v'))` to get the "current visual selection":
vim.keymap.set('x', 'M', function()
  local region = vim.fn.getregionpos(vim.fn.getpos('v'), vim.fn.getpos('.'), {
    type = 'v',
    exclusive = false,
    eol = false,
    bounds = true,
  })
  local line1 = region[1][1][2]
  local line2 = region[1][2][2]
  vim.print({ line1, line2 })
end)

vim.keymap.set({ 'n', 'i' }, { 'a', 'b' }, '<cmd>echom localtime()<cr>')
-- ... is the same as:
vim.keymap.set('n', 'a', '<cmd>echom localtime()<cr>')
vim.keymap.set('i', 'a', '<cmd>echom localtime()<cr>')
vim.keymap.set('n', 'b', '<cmd>echom localtime()<cr>')
vim.keymap.set('i', 'b', '<cmd>echom localtime()<cr>')
```

Parameters:
- {modes} (`string|string[]`) Mode "short-name" (see nvim_set_keymap()), or a list thereof.
- {lhs} (`string|string[]`) Left-hand side {lhs} of the mapping, or a list thereof.
- {rhs} (`string|function`) Right-hand side {rhs} of the mapping, can be a Lua function.
- {opts} (`table?`) Table of :map-arguments. Same as nvim_set_keymap() {opts}, except:
  - {replace_keycodes} defaults to `true` if "expr" is `true`.
  - {noremap} is not supported; use {remap} instead (see below).

Also accepts:
- {buf}? (`integer`) Creates buffer-local mapping, `0` for current buffer.
- {remap}? (`boolean`, default: `false`) Make the mapping recursive. Inverse of {noremap}.

See also:
- nvim_set_keymap()
- maparg()
- mapcheck()
- mapset()

Lua module: vim.loader

vim.loader

vim.loader.enable({enable}) vim.loader.enable()

WARNING: This feature is experimental/unstable.

Enables or disables the experimental Lua module loader:

Enable (`enable=true`):
- overrides loadfile()
- adds the Lua loader using the byte-compilation cache
- adds the libs loader
- removes the default Nvim loader

Disable (`enable=false`):
- removes the loaders
- adds the default Nvim loader

Parameters:
- {enable} (`boolean?`) true/nil to enable, false to disable

vim.loader.find({modname}, {opts}) vim.loader.find()

WARNING: This feature is experimental/unstable.

Finds Lua modules for the given module name.

Parameters:
- {modname} (`string`) Module name, or `"*"` to find the top-level modules instead
- {opts} (`table?`) Options for finding a module:
  - {all}? (`boolean`, default: `false`) Search for all matches.
  - {paths}? (`string[]`, default: {}) Extra paths to search for modname
  - {patterns}? (`string[]`, default: `{"/init.lua", ".lua"}`) List of patterns to use when searching for modules. A pattern is a string added to the basename of the Lua module being searched.
  - {rtp}? (`boolean`, default: `true`) Search for modname in the runtime path.

Return:
(`table[]`) A list of objects with the following fields:
- {modname} (`string`) Name of the module

- {modpath} (string) Path of the module
- {stat}? (uv.fs_stat.result) The fs_stat of the module path. Won't be returned for modname="*"

vim.loader.reset({path}) vim.loader.reset()
WARNING: This feature is experimental/unstable.

Resets the cache for the path, or all the paths if path is nil.

Parameters:
- {path} (string?) path to reset

Lua module: vim.log vim.log

The vim.log module provides file-backed logger instances.

Use vim.log.new() to create a logger, it exposes five writer functions: trace(), debug(), info(), warn(), and error(). They are correspond to log levels defined by vim.log.levels.

Example:

```
local log = vim.log.new({ name = 'my-plugin', })
-- By default, logs will be written to {name}.log under `stdpath('log')`.

log.error('request failed', 'timeout')
-- This will write a line like the following to the log file:
--   [ERROR][2024-01-01 12:00:00] source.lua:123    request failed    timeout

-- Set the log level to `INFO`, otherwise the `info()` call below would be ignored,
-- since the default level is `WARN`.
vim.log.set_level(log, vim.log.levels.INFO)
log.info('starting', { buf = vim.api.nvim_get_current_buf() })
-- This will write a line like the following to the log file:
--   [INFO][2024-01-01 12:01:00] source.lua:124    starting    { buf = 1 }
```

You can also provide a custom formatter function to customize the log output format.

vim.Log

Fields:
- {debug} (fun(...:any): boolean?) Writes a message at vim.log.levels.DEBUG.
- {error} (fun(...:any): boolean?) Writes a message at vim.log.levels.ERROR.
- {info} (fun(...:any): boolean?) Writes a message at vim.log.levels.INFO.
- {trace} (fun(...:any): boolean?) Writes a message at vim.log.levels.TRACE.
- {warn} (fun(...:any): boolean?) Writes a message at vim.log.levels.WARN.

get_level({log}) vim.log.get_level()
Gets the current log level.

Parameters:
- {log} (vim.Log)

Return:
(vim.log.levels)

new({opts}) vim.log.new()
Creates a logger instance.

The logger writes formatted messages to a file, using a per-instance log level and formatting function.

Parameters:
- {opts} (table) A table with the following fields:
  - {format_func}? (fun(min_level: vim.log.levels, level: vim.log.levels, ...): string?) Formatter used for each log entry. Receives the logger's minimum level, the message level, and the values passed to the writer. Return a string to write an entry, or nil to skip it.
  - {level}? (vim.log.levels, default: vim.log.levels.WARN) Minimum level that will be written.
  - {name} (string) Display name used in notifications emitted by the logger.

Return:
(vim.Log)

set_format_func({log}, {func}) vim.log.set_format_func()
Sets the formatter used to produce log entries.

The formatter receives the logger's minimum level, the message level, and the values passed to the writer method. Return a string to write an entry, or nil to skip it.

Parameters:
- {log} (vim.Log)
- {func} (fun(min_level: vim.log.levels, level: vim.log.levels, ...): string?)

set_level({log}, {level}) vim.log.set_level()
Sets the current log level.

Entries below this level are skipped.

Parameters:
- {log} (vim.Log)
- {level} (vim.log.levels)

Lua module: vim.lpeg

vim.lpeg

LPeg is a pattern-matching library for Lua, based on Parsing Expression Grammars (PEGs). https://bford.info/packrat/

lua-lpeg vim.lpeg.Pattern The LPeg library for parsing expression grammars is included as `vim.lpeg` (https://www.inf.puc-rio.br/~roberto/lpeg/).

In addition, its regex-like interface is available as vim.re (https://www.inf.puc-rio.br/~roberto/lpeg/re.html).

Pattern:match({subject}, {init}, {...}) Pattern:match()

Matches the given `pattern` against the `subject` string. If the match succeeds, returns the index in the subject of the first character after the match, or the captured values (if the pattern captured any value). An optional numeric argument `init` makes the match start at that position in the subject string. As usual in Lua libraries, a negative value counts from the end. Unlike typical pattern-matching functions, `match` works only in anchored mode; that is, it tries to match the pattern with a prefix of the given subject string (at position `init`), not with an arbitrary substring of the subject. So, if we want to find a pattern anywhere in a string, we must either write a loop in Lua or write a pattern that matches anywhere.

Example:

```
local pattern = lpeg.R('az') ^ 1 * -1
assert(pattern:match('hello') == 6)
assert(lpeg.match(pattern, 'hello') == 6)
assert(pattern:match('1 hello') == nil)
```

Parameters:

- `{subject}` (`string`)
- `{init}` (`integer?`)
- `{...}` (`any`)

Return:

(`any`) ...

vim.lpeg.B({pattern}) vim.lpeg.B()

Returns a pattern that matches only if the input string at the current position is preceded by `patt`. Pattern `patt` must match only strings with some fixed length, and it cannot contain captures. Like the and predicate, this pattern never consumes any input, independently of success or failure.

Parameters:

- `{pattern}` (`vim.lpeg.Pattern|string|integer|boolean|table`)

Return:

(`vim.lpeg.Pattern`)

vim.lpeg.C({patt}) vim.lpeg.C()

Creates a simple capture, which captures the substring of the subject that matches `patt`. The captured value is a string. If `patt` has other captures, their values are returned after this one.

Example:

```
local function split (s, sep)
  sep = lpeg.P(sep)
  local elem = lpeg.C((1 - sep) ^ 0)
  local p = elem * (sep * elem) ^ 0
  return lpeg.match(p, s)
end
local a, b, c = split('a,b,c', ',')
assert(a == 'a')
assert(b == 'b')
assert(c == 'c')
```

Parameters:

- `{patt}` (`vim.lpeg.Pattern|string|integer|boolean|table|function`)

Return:

(`vim.lpeg.Capture`)

vim.lpeg.Carg({n}) vim.lpeg.Carg()

Creates an argument capture. This pattern matches the empty string and produces the value given as the nth extra argument given in the call to `lpeg.match`.

Parameters:

- `{n}` (`integer`)

Return:

(`vim.lpeg.Capture`)

vim.lpeg.Cb({name}) vim.lpeg.Cb()

Creates a back capture. This pattern matches the empty string and produces the values produced by the most recent group capture named `name` (where `name` can be any Lua value). Most recent means the last complete outermost group capture with the given name. A Complete capture means that the entire pattern corresponding to the capture has matched. An Outermost capture means that the capture is not inside another complete capture. In the same way that LPeg does not specify when it evaluates captures, it does not specify whether it reuses values previously produced by the group or re-evaluates them.

Parameters:

- `{name}` (`any`)

Return:

(`vim.lpeg.Capture`)

vim.lpeg.Cc({...}) vim.lpeg.Cc()
Creates a constant capture. This pattern matches the empty string and produces all given values as its captured values.

Parameters:
- {...} (any)

Return:
(vim.lpeg.Capture)

vim.lpeg.Cf({patt}, {func}) vim.lpeg.Cf()
Creates a fold capture. If patt produces a list of captures C1 C2 ... Cn, this capture will produce the value func(...func(func(C1, C2), C3)...,Cn), that is, it will fold (or accumulate, or reduce) the captures from patt using function func. This capture assumes that patt should produce at least one capture with at least one value (of any type), which becomes the initial value of an accumulator. (If you need a specific initial value, you may prefix a constant capture to patt.) For each subsequent capture, LPeg calls func with this accumulator as the first argument and all values produced by the capture as extra arguments; the first result from this call becomes the new value for the accumulator. The final value of the accumulator becomes the captured value.

Example:

```
local number = lpeg.R('09') ^ 1 / tonumber
local list = number * (',' * number) ^ 0
local function add(acc, newvalue) return acc + newvalue end
local sum = lpeg.Cf(list, add)
assert(sum:match('10,30,43') == 83)
```

Parameters:
- {patt} (vim.lpeg.Pattern|string|integer|boolean|table|function)
- {func} (fun(acc, newvalue))

Return:
(vim.lpeg.Capture)

vim.lpeg.Cg({patt}, {name}) vim.lpeg.Cg()
Creates a group capture. It groups all values returned by patt into a single capture. The group may be anonymous (if no name is given) or named with the given name (which can be any non-nil Lua value).

Parameters:
- {patt} (vim.lpeg.Pattern|string|integer|boolean|table|function)
- {name} (string?)

Return:
(vim.lpeg.Capture)

vim.lpeg.Cmt({patt}, {fn}) vim.lpeg.Cmt()
Creates a match-time capture. Unlike all other captures, this one is evaluated immediately when a match occurs (even if it is part of a larger pattern that fails later). It forces the immediate evaluation of all its nested captures and then calls function. The given function gets as arguments the entire subject, the current position (after the match of patt), plus any capture values produced by patt. The first value returned by function defines how the match happens. If the call returns a number, the match succeeds and the returned number becomes the new current position. (Assuming a subject sand current position i, the returned number must be in the range [i, len(s) + 1].) If the call returns true, the match succeeds without consuming any input (so, to return true is equivalent to return i). If the call returns false, nil, or no value, the match fails. Any extra values returned by the function become the values produced by the capture.

Parameters:
- {patt} (vim.lpeg.Pattern|string|integer|boolean|table|function)
- {fn} (fun(s: string, i: integer, ...: any)) (position: boolean|integer, ...: any)

Return:
(vim.lpeg.Capture)

vim.lpeg.Cp() vim.lpeg.Cp()
Creates a position capture. It matches the empty string and captures the position in the subject where the match occurs. The captured value is a number.

Example:

```
local I = lpeg.Cp()
local function anywhere(p) return lpeg.P({I * p * I + 1 * lpeg.V(1)}) end
local match_start, match_end = anywhere('world'):match('hello world!')
assert(match_start == 7)
assert(match_end == 12)
```

Return:
(vim.lpeg.Capture)

vim.lpeg.Cs({patt}) vim.lpeg.Cs()
Creates a substitution capture. This function creates a substitution capture, which captures the substring of the subject that matches patt, with substitutions. For any capture inside patt with a value, the substring that matched the capture is replaced by the capture value (which should be a string). The final captured value is the string resulting from all replacements.

Example:

```
local function gsub (s, patt, repl)
  patt = lpeg.P(patt)
  patt = lpeg.Cs((patt / repl + 1) ^ 0)
  return lpeg.match(patt, s)
end
assert(gsub('Hello, xxx!', 'xxx', 'World') == 'Hello, World!')
```

Parameters:
- {patt} (vim.lpeg.Pattern|string|integer|boolean|table|function)

Return:
(vim.lpeg.Capture)

vim.lpeg.Ct({patt}) vim.lpeg.Ct()
Creates a table capture. This capture returns a table with all values from all anonymous captures made by patt inside this table in successive integer keys, starting at 1. Moreover, for each named capture group created by patt, the first value of the group is put into the table with the group name as its key. The captured value is only the table.

Parameters:
- {patt} (vim.lpeg.Pattern|string|integer|boolean|table|function)

Return:
(vim.lpeg.Capture)

vim.lpeg.locale({tab}) vim.lpeg.locale()
Returns a table with patterns for matching some character classes according to the current locale. The table has fields named alnum, alpha, cntrl, digit, graph, lower, print, punct, space, upper, and xdigit, each one containing a correspondent pattern. Each pattern matches any single character that belongs to its class. If called with an argument table, then it creates those fields inside the given table and returns that table.

Example:

```
lpeg.locale(lpeg)
local space = lpeg.space ^ 0
local name = lpeg.C(lpeg.alpha ^ 1) * space
local sep = lpeg.S(',;') * space
local pair = lpeg.Cg(name * '=' * space * name) * sep ^ -1
local list = lpeg.Cf(lpeg.Ct('') * pair ^ 0, rawset)
local t = list:match('a=b, c = hi; next = pi')
assert(t.a == 'b')
assert(t.c == 'hi')
assert(t.next == 'pi')
local locale = lpeg.locale()
assert(type(locale.digit) == 'userdata')
```

Parameters:
- {tab} (table?)

Return:
(vim.lpeg.Locale)

vim.lpeg.match({pattern}, {subject}, {init}, {...}) vim.lpeg.match()
Matches the given pattern against the subject string. If the match succeeds, returns the index in the subject of the first character after the match, or the captured values (if the pattern captured any value). An optional numeric argument init makes the match start at that position in the subject string. As usual in Lua libraries, a negative value counts from the end. Unlike typical pattern-matching functions, match works only in anchored mode; that is, it tries to match the pattern with a prefix of the given subject string (at position init), not with an arbitrary substring of the subject. So, if we want to find a pattern anywhere in a string, we must either write a loop in Lua or write a pattern that matches anywhere.

Example:

```
local pattern = lpeg.R('az') ^ 1 * -1
assert(pattern:match('hello') == 6)
assert(lpeg.match(pattern, 'hello') == 6)
assert(pattern:match('1 hello') == nil)
```

Parameters:
- {pattern} (vim.lpeg.Pattern|string|integer|boolean|table|function)
- {subject} (string)
- {init} (integer?)
- {...} (any)

Return:
(any) ...

vim.lpeg.P({value}) vim.lpeg.P()
Converts the given value into a proper pattern. The following rules are applied:
- If the argument is a pattern, it is returned unmodified.
- If the argument is a string, it is translated to a pattern that matches the string literally.
- If the argument is a non-negative number n, the result is a pattern that matches exactly n characters.
- If the argument is a negative number -n, the result is a pattern that succeeds only if the input string has less than n characters left: lpeg.P(-n) is equivalent to -lpeg.P(n) (see the unary minus operation).
- If the argument is a boolean, the result is a pattern that always succeeds or always fails (according to the boolean value), without consuming any input.
- If the argument is a table, it is interpreted as a grammar (see Grammars).
- If the argument is a function, returns a pattern equivalent to a match-time capture over the empty string.

Parameters:
- {value} (vim.lpeg.Pattern|string|integer|boolean|table|function)

Return:
(vim.lpeg.Pattern)

vim.lpeg.R({...}) vim.lpeg.R()
Returns a pattern that matches any single character belonging to one of the given ranges. Each range is a string xy of length 2, representing

all characters with code between the codes of x and y (both inclusive). As an example, the pattern `lpeg.R('09')` matches any digit, and `lpeg.R('az', 'AZ')` matches any ASCII letter.

Example:

```
local pattern = lpeg.R('az') ^ 1 * -1
assert(pattern:match('hello') == 6)
```

Parameters:
- `{...}` (`string`)

Return:
(`vim.lpeg.Pattern`)

vim.lpeg.S({`string`}) vim.lpeg.S()
Returns a pattern that matches any single character that appears in the given string (the S stands for Set). As an example, the pattern `lpeg.S('+-*/')` matches any arithmetic operator. Note that, if `s` is a character (that is, a string of length 1), then `lpeg.P(s)` is equivalent to `lpeg.S(s)` which is equivalent to `lpeg.R(s..s)`. Note also that both `lpeg.S('')` and `lpeg.R()` are patterns that always fail.

Parameters:
- `{string}` (`string`)

Return:
(`vim.lpeg.Pattern`)

vim.lpeg.setmaxstack({`max`}) vim.lpeg.setmaxstack()
Sets a limit for the size of the backtrack stack used by LPeg to track calls and choices. The default limit is `400`. Most well-written patterns need little backtrack levels and therefore you seldom need to change this limit; before changing it you should try to rewrite your pattern to avoid the need for extra space. Nevertheless, a few useful patterns may overflow. Also, with recursive grammars, subjects with deep recursion may also need larger limits.

Parameters:
- `{max}` (`integer`)

vim.lpeg.type({`value`}) vim.lpeg.type()
Returns the string `"pattern"` if the given value is a pattern, otherwise `nil`.

Parameters:
- `{value}` (`vim.lpeg.Pattern|string|integer|boolean|table|function`)

Return:
(`"pattern"?`)

vim.lpeg.V({`v`}) vim.lpeg.V()
Creates a non-terminal (a variable) for a grammar. This operation creates a non-terminal (a variable) for a grammar. The created non-terminal refers to the rule indexed by `v` in the enclosing grammar.

Example:

```
local b = lpeg.P({'(' * ((1 - lpeg.S '()') + lpeg.V(1)) ^ 0 * ')'})
assert(b:match('((string))') == 11)
assert(b:match('(') == nil)
```

Parameters:
- `{v}` (`boolean|string|number|function|table|thread|userdata|lightuserdata`)

Return:
(`vim.lpeg.Pattern`)

vim.lpeg.version() vim.lpeg.version()
Returns a string with the running version of LPeg.

Return:
(`string`)

Lua module: vim.mpack vim.mpack

This module provides encoding and decoding of Lua objects to and from msgpack-encoded strings. Supports vim.NIL and vim.empty_dict().

vim.mpack.decode({`str`}) vim.mpack.decode()
Decodes (or "unpacks") the msgpack-encoded `{str}` to a Lua object.

Parameters:
- `{str}` (`string`)

Return:
(`any`)

vim.mpack.encode({`obj`}) vim.mpack.encode()
Encodes (or "packs") Lua object `{obj}` as msgpack in a Lua string.

Parameters:
- `{obj}` (`any`)

Return:
(`string`)

Lua module: vim.net vim.net

vim.net.request.Response

Fields:
- {body} (string) The HTTP body of the request

vim.net.request()

vim.net.request({method}, {url}, {opts}, {on_response}) Makes an HTTP request to the given URL, asynchronously passing the result to the specified on_response, outpath or outbuf.

Examples:

```
-- Write response body to file.
vim.net.request('https://neovim.io/charter/', {
  outpath = 'vision.html',
})

-- Process the response.
vim.net.request(
  'https://api.github.com/repos/neovim/neovim',
  {},
  function (err, res)
    if err then return end
    local stars = vim.json.decode(res.body).stargazers_count
    vim.print(('Neovim currently has %d stars'):format(stars))
  end
)

-- Write to both file and current buffer, but cancel it.
local job = vim.net.request('https://neovim.io/charter/', {
  outpath = 'vision.html',
  outbuf = 0,
})
job:close()

-- Add custom headers in the request.
vim.net.request('https://neovim.io/charter/', {
  headers = { Authorization = 'Bearer XYZ' },
})

-- POST request with body.
vim.net.request('POST', 'https://example.com/api', {
  body = '{"key": "value"}',
  headers = {['Content-Type'] = 'application/json' }
})
```

Parameters:
- {method} (vim.net.HttpMethod) (default: GET) The HTTP method (GET, POST, PUT, PATCH, HEAD, DELETE).
- {url} (string) The URL for the request.
- {opts} (table?) A table with the following fields:
  - {body}? (string) Request body for POST/PUT/PATCH requests.
  - {headers}? (table<string, string>) Custom headers to send with the request. Supports basic key/value headers and empty headers as supported by curl. Does not support "@filename" style, internal header deletion ("Header:").
  - {outbuf}? (integer) Buffer to save the response body to.
  - {outpath}? (string) File path to save the response body to.
  - {retry}? (integer) Number of retries on transient failures (default: 3).
  - {verbose}? (boolean) Enables verbose output.
- {on_response} (vim.net.request.ResponseFunc?) Callback invoked on request completion.

Overloads:
- fun(url: string, opts: vim.net.request.Opts, response: vim.net.request.ResponseFunc)
- fun(method: vim.net.HttpMethod, url: string, opts: vim.net.request.Opts, response: vim.net.request.ResponseFunc)

Return:
({ close: fun() }) Object with close() method which cancels the request.

Lua module: vim.pos

vim.pos

EXPERIMENTAL: This API is unstable, do not use it. Its semantics are not yet finalized. Subscribe to this issue to stay updated: https://github.com/neovim/neovim/issues/25509

Provides operations to compare, calculate, and convert positions represented by vim.Pos objects.

vim.Pos Represents a buffer position based on api-indexing (0-indexed, end-exclusive ranges). Call vim.pos() to create a new vim.Pos by passing the {buf}, {row}, and {col} of a position.

Example:

```
local pos1 = vim.pos(0, 3, 5)
local pos2 = vim.pos(0, 4, 0)

-- Operators are overloaded for comparing two `vim.Pos` objects.
if pos1 < pos2 then
  print("pos1 comes before pos2")
end

if pos1 ~= pos2 then
  print("pos1 and pos2 are different positions")
end
```

It may include optional fields that enable additional capabilities, such as format conversions.

Fields:
- {buf} (integer) buffer handle.
- {col} (integer) 0-based byte index.
- {row} (integer) 0-based byte index.

cursor({buf}, {pos}) vim.pos.cursor()
Creates a new vim.Pos from cursor position (see api-indexing).

If {pos} is omitted, the first argument is treated as {win} instead of {buf}, and the current cursor position of {win} is used. If {win} is also omitted, it defaults to the current window.

Example:

```
local buf = vim.api.nvim_win_get_buf(0)
local cursor_pos = vim.api.nvim_win_get_cursor(0)
local pos = vim.pos.cursor(buf, cursor_pos)
-- This is equivalent:
local pos = vim.pos.cursor(0)
```

Parameters:
- {buf} (integer)
- {pos} ([integer, integer]) (lnum, col) tuple

Overloads:
- fun(win?: integer): vim.Pos

Return:
(vim.Pos)

extmark({buf}, {row}, {col}) vim.pos.extmark()
Creates a new vim.Pos from extmark position (see api-indexing).

Example:

```
local pos = vim.pos.extmark(0, 3, 5)
```

Parameters:
- {buf} (integer)
- {row} (integer)
- {col} (integer)

Return:
(vim.Pos)

lsp({buf}, {pos}, {position_encoding}) vim.pos.lsp()
Creates a new vim.Pos from lsp.Position.

Example:

```
local lsp_pos = {
  line = 3,
  character = 5
}

local pos = vim.pos.lsp(0, lsp_pos, 'utf-16')
```

Parameters:
- {buf} (integer)
- {pos} (lsp.Position)
- {position_encoding} (lsp.PositionEncodingKind)

Return:
(vim.Pos)

mark({buf}, {lnum}, {col}) vim.pos.mark()
Creates a new vim.Pos from mark position (see api-indexing).

Example:

```
local mark_info = vim.api.nvim_get_mark('M')
local lnum, col, buf, name = unpack(mark_info)

if lnum == 0 and col == 0 and buf == 0 then
  return -- mark 'M' is not set.
end

local pos = vim.pos.mark(buf, lnum, col)
```

Parameters:
- {buf} (integer)
- {lnum} (integer)
- {col} (integer)

Return:
(vim.Pos)

offset({buf}, {offset}) vim.pos.offset()
Creates a new vim.Pos from buffer (bytes) offset.

Example:

```
local offset = vim.api.nvim_buf_get_offset(0, vim.api.nvim_buf_line_count(0))
local pos = vim.pos.offset(0, offset)
```

Parameters:
- {buf} (integer)
- {offset} (integer)

Return:
(vim.Pos)

to_cursor({pos}) vim.pos.to_cursor()
Converts vim.Pos to cursor position (see api-indexing).

Example:

```
local pos = vim.pos(0, 3, 5)

-- Convert to cursor position, you can call it in a method style.
local cursor_pos = pos:to_cursor()
vim.api.nvim_win_set_cursor(0, cursor_pos)
```

Parameters:
- {pos} (vim.Pos)

Return:
([integer, integer]) (lnum, col) tuple

to_extmark({pos}) vim.pos.to_extmark()
Converts vim.Pos to extmark position (see api-indexing).

Example:

```
local pos = vim.pos(0, 3, 5)

-- Convert to extmark position, you can call it in a method style.
local extmark_pos = pos:to_extmark()
```

Parameters:
- {pos} (vim.Pos)

Return (multiple):
(integer) row (integer) col

to_lsp({pos}, {position_encoding}) vim.pos.to_lsp()
Converts vim.Pos to lsp.Position.

Example:

```
local pos = vim.pos(0, 3, 5)

-- Convert to LSP position, you can call it in a method style.
local lsp_pos = pos:to_lsp('utf-16')
```

Parameters:
- {pos} (vim.Pos)
- {position_encoding} (lsp.PositionEncodingKind)

Return:
(lsp.Position)

to_mark({pos}) vim.pos.to_mark()
Converts vim.Pos to mark position (see api-indexing).

Example:

```
local pos = vim.pos(0, 3, 5)

-- Convert to mark position, you can call it in a method style.
local lnum, col = pos:to_mark()
vim.api.nvim_buf_set_mark(0, 'M', lnum, col)
```

Parameters:
- {pos} (vim.Pos)

Return (multiple):
(integer) lnum (integer) col

to_offset({pos}) vim.pos.to_offset()
Converts vim.Pos to buffer (bytes) offset.

Example:

```
local p1 = vim.pos(0, 3, 5)
local p2 = vim.pos(0, 4, 0)

-- Convert to buffer offset, you can call it in a method style.
local offset1 = p1:to_offset()
local offset2 = p2:to_offset()
-- Can be used to calculate the distance between two locations.
local distance = offset2 - offset1
```

Parameters:
- {pos} (vim.Pos)

Return:
(integer)

Lua module: vim.range

vim.range

EXPERIMENTAL: This API is unstable, do not use it. Its semantics are not yet finalized. Subscribe to this issue to stay updated: https://github.com/neovim/neovim/issues/25509

Provides operations to compare, calculate, and convert ranges represented by vim.Range objects.

vim.Range Represents a range based on api-indexing (0-indexed, end-exclusive). Call `vim.range()` to create a new range by passing start and end positions (vim.Pos).

Both positions must have the same optional fields, which may enable additional capabilities (such as format conversions).

Example:

```
local pos1 = vim.pos(0, 3, 5)
local pos2 = vim.pos(0, 4, 0)

-- Create a range from two positions.
local range1 = vim.range(pos1, pos2)
-- Or create a range from four integers representing start and end positions.
local range2 = vim.range(0, 3, 5, 4, 0)

-- Because `vim.Range` is end exclusive, `range1` and `range2` both represent
-- a range starting at the row 3, column 5 and ending at where the row 3 ends
-- (including the newline at the end of line 3).

-- Operators are overloaded for comparing two `vim.Pos` objects.
if range1 == range2 then
  print("range1 and range2 are the same range")
end
```

Fields:
- {buf} (integer) Optional buffer handle.
- {end_col} (integer) 0-based byte index.
- {end_row} (integer) 0-based byte index.
- {start_col} (integer) 0-based byte index.
- {start_row} (integer) 0-based byte index.

vim.range.extmark()

extmark({buf}, {start_row}, {start_col}, {end_row}, {end_col}) Creates a new vim.Range from extmark range (see api-indexing).

Example:

```
local range = vim.range.extmark(0, 3, 5, 4, 0)
```

Parameters:
- {buf} (integer)
- {start_row} (integer)
- {start_col} (integer)
- {end_row} (integer)
- {end_col} (integer)

Return:
(vim.Range)

has({outer}, {inner}) vim.range.has()

Checks whether {outer} range contains {inner} range or position.

Parameters:
- {outer} (vim.Range)
- {inner} (vim.Range|vim.Pos)

Return:
(boolean) `true` if {outer} range fully contains {inner} range or position.

intersect({r1}, {r2}) vim.range.intersect()

Computes the common range shared by the given ranges.

Parameters:
- {r1} (vim.Range) First range to intersect.
- {r2} (vim.Range) Second range to intersect

Return:
(vim.Range?) range that is present inside both `r1` and `r2`. `nil` if such range does not exist.

is_empty({range}) vim.range.is_empty()
Checks whether the given range is empty; i.e., start >= end.

Parameters:
- {range} (vim.Range)

Return:
(boolean) true if the given range is empty.

lsp({buf}, {range}, {position_encoding}) vim.range.lsp()
Creates a new vim.Range from lsp.Range.

Example:

```
local lsp_range = {
  ['start'] = { line = 3, character = 5 },
  ['end'] = { line = 4, character = 0 }
}

local range = vim.range.lsp(0, lsp_range, 'utf-16')
```

Parameters:
- {buf} (integer)
- {range} (lsp.Range)
- {position_encoding} (lsp.PositionEncodingKind)

Return:
(vim.Range)

vim.range.mark()
mark({buf}, {start_lnum}, {start_col}, {end_lnum}, {end_col}) Creates a new vim.Range from "mark-indexed" range (see api-indexing).

Example:

```
-- A range represented by marks may be end-inclusive (decided by 'selection' option).
local start_lnum, start_col = unpack(api.nvim_buf_get_mark(bufnr, '<'))
local end_lnum, end_col = unpack(api.nvim_buf_get_mark(bufnr, '>'))

-- Create an end-exclusive range.
local range = vim.range.mark(0, start_lnum, start_col, end_lnum, end_col)
```

Parameters:
- {buf} (integer)
- {start_lnum} (integer)
- {start_col} (integer)
- {end_lnum} (integer)
- {end_col} (integer)

Return:
(vim.Range)

to_extmark({range}) vim.range.to_extmark()
Converts vim.Range to extmark range (see api-indexing).

Example:

```
local range = vim.range(0, 3, 5, 4, 0)

-- Convert to extmark range, you can call it in a method style.
local extmark_range = range:to_extmark()
```

Parameters:
- {range} (vim.Range)

Return (multiple):
(integer) start_row (integer) start_col (integer) end_row (integer) end_col

to_lsp({range}, {position_encoding}) vim.range.to_lsp()
Converts vim.Range to lsp.Range.

Example:

```
local range = vim.range(0, 3, 5, 4, 0)

-- Convert to LSP range, you can call it in a method style.
local lsp_range = range:to_lsp('utf-16')
```

Parameters:
- {range} (vim.Range)
- {position_encoding} (lsp.PositionEncodingKind)

Return:
(lsp.Range)

to_mark({range}) vim.range.to_mark()
Converts vim.Range to extmark range (see api-indexing).

Example:

```
local range = vim.range(0, 3, 5, 4, 0)

-- Convert to mark range, you can call it in a method style.
local start_lnum, start_col, end_lnum, end_col = range:to_mark()
```

Parameters:
- {range} (vim.Range)

Return (multiple):
(integer) start_lnum (integer) start_col (integer) end_lnum (integer) end_col

Lua module: vim.re

vim.re

The vim.re module provides a conventional regex-like syntax for pattern usage within LPeg vim.lpeg. (Unrelated to vim.regex which provides Vim regexp from Lua.)

See https://www.inf.puc-rio.br/~roberto/lpeg/re.html for the original documentation including regex syntax and examples.

vim.re.compile({string}, {defs}) vim.re.compile()
Compiles the given {string} and returns an equivalent LPeg pattern. The given string may define either an expression or a grammar. The optional {defs} table provides extra Lua values to be used by the pattern.

Parameters:
- {string} (string)
- {defs} (table?)

Return:
(vim.lpeg.Pattern)

vim.re.find({subject}, {pattern}, {init}) vim.re.find()
Searches the given {pattern} in the given {subject}. If it finds a match, returns the index where this occurrence starts and the index where it ends. Otherwise, returns nil.

An optional numeric argument {init} makes the search starts at that position in the subject string. As usual in Lua libraries, a negative value counts from the end.

Parameters:
- {subject} (string)
- {pattern} (vim.lpeg.Pattern|string)
- {init} (integer?)

Return (multiple):
(integer?) the index where the occurrence starts, nil if no match (integer?) the index where the occurrence ends, nil if no match

vim.re.gsub({subject}, {pattern}, {replacement}) vim.re.gsub()
Does a global substitution, replacing all occurrences of {pattern} in the given {subject} by {replacement}.

Parameters:
- {subject} (string)
- {pattern} (vim.lpeg.Pattern|string)
- {replacement} (string)

Return:
(string)

vim.re.match({subject}, {pattern}, {init}) vim.re.match()
Matches the given {pattern} against the given {subject}, returning all captures.

Parameters:
- {subject} (string)
- {pattern} (vim.lpeg.Pattern|string)
- {init} (integer?)

Return:
(integer|vim.lpeg.Capture?)

See also:
- vim.lpeg.match()

vim.re.updatelocale() vim.re.updatelocale()
Updates the pre-defined character classes to the current locale.

Lua module: vim.regex

vim.regex

The vim.regex interface allows Lua code to use Vim regexes.
- Matches within a single line.
- Does not check for interrupts (CTRL-C), so can be used from an api-fast context.

regex:match_line()
regex:match_line({buf}, {line_idx}, {start}, {end_}) Matches line at line_idx (zero-based) in buffer buf. Match is restricted to byte index range start and end_ if given, otherwise see regex:match_str(). Returned byte indices are relative to start if given.

Parameters:
- {buf} (integer)
- {line_idx} (integer)
- {start} (integer?)
- {end_} (integer?)

Return (multiple):
(`integer?`) match start (byte index) relative to `start`, or `nil` if no match (`integer?`) match end (byte index) relative to `start`, or `nil` if no match

regex:match_str({str}) regex:match_str()
Matches string `str` against this regex. To match the string precisely, surround the regex with "^" and "$". Returns the byte indices for the start and end of the match, or `nil` if there is no match. Because any integer is "truthy", `regex:match_str()` can be directly used as a condition in an if-statement.

Parameters:
- `{str}` (`string`)

Return (multiple):
(`integer?`) match start (byte index), or `nil` if no match (`integer?`) match end (byte index), or `nil` if no match

vim.regex({re}) vim.regex()
Parses the Vim regex `re` and returns a regex object. Regexes are "magic" and case-sensitive by default, regardless of 'magic' and 'ignorecase'. They can be controlled with flags, see /magic and /ignorecase.

Parameters:
- `{re}` (`string`)

Return:
(`vim.regex`)

Lua module: vim.secure vim.secure

vim.secure.read({path}) vim.secure.read()
If `{path}` is a file: attempt to read the file, prompting the user if the file should be trusted.

If `{path}` is a directory: return true if the directory is trusted (non-recursive), prompting the user as necessary.

The user's choice is persisted in a trust database at $XDG_STATE_HOME/nvim/trust.

Attributes:
Since: 0.9.0

Parameters:
- `{path}` (`string`) Path to a file or directory to read.

Return:
(`boolean|string?`) If `{path}` is not trusted or does not exist, returns `nil`. Otherwise, returns the contents of `{path}` if it is a file, or true if `{path}` is a directory.

See also:
- :trust

vim.secure.trust({opts}) vim.secure.trust()
Manage the trust database.

The trust database is located at $XDG_STATE_HOME/nvim/trust.

Attributes:
Since: 0.9.0

Parameters:
- `{opts}` (`table`) A table with the following fields:
  - `{action}` (`'allow'|'deny'|'remove'`) One of:
    - `'allow'` to add a file to the trust database and trust it,
    - `'deny'` to add a file to the trust database and deny it,
    - `'remove'` to remove file from the trust database
  - `{bufnr}?` (`integer`) Buffer number to update. Mutually exclusive with `{path}`.
  - `{path}?` (`string`) Path to a file to update. Mutually exclusive with `{bufnr}`.

Return (multiple):
(`boolean`) success true if operation was successful (`string`) msg full path if operation was successful, else error message

Lua module: vim.snippet vim.snippet

Snippet expansion and navigation.

Internal autocmds live in the `nvim.snippet` augroup. To disable them (e.g. to manage sessions yourself), clear it:

```
vim.api.nvim_clear_autocmds({ group = 'nvim.snippet', buffer = 0 })
```

Heads-up: vim.snippet.jump() re-registers them on every jump.

vim.snippet.ActiveFilter

Fields:
- `{direction}` (`vim.snippet.Direction`) Navigation direction. -1 for previous, 1 for next.

vim.snippet.active({filter}) vim.snippet.active()
Returns `true` if there's an active snippet in the current buffer, applying the given filter if provided.

Parameters:
- `{filter}` (`vim.snippet.ActiveFilter?`) Filter to constrain the search with:
  - `direction` (vim.snippet.Direction): Navigation direction. Will return `true` if the snippet can be jumped in the given direction.

Return:

(boolean)

vim.snippet.expand({input}) vim.snippet.expand()
Expands the given snippet text. Refer to https://microsoft.github.io/language-server-protocol/specification/#snippet_syntax for the specification of valid input.

Tabstops are highlighted with hl-SnippetTabstop and hl-SnippetTabstopActive.

Parameters:
- {input} (string)

vim.snippet.jump({direction}) vim.snippet.jump()
Jumps to the next (or previous) placeholder in the current snippet, if possible.

By default <Tab> is setup to jump if a snippet is active. The default mapping looks like:

```
vim.keymap.set({ 'i', 's' }, '<Tab>', function()
   if vim.snippet.active({ direction = 1 }) then
     return '<Cmd>lua vim.snippet.jump(1)<CR>'
   else
     return '<Tab>'
   end
 end, { desc = '...', expr = true, silent = true })
```

Parameters:
- {direction} (vim.snippet.Direction) Navigation direction. -1 for previous, 1 for next.

vim.snippet.stop() vim.snippet.stop()
Exits the current snippet.

Lua module: vim.spell vim.spell

vim.spell.check({str}) vim.spell.check()
Check {str} for spelling errors. Similar to the Vimscript function spellbadword().

Note: The behaviour of this function is dependent on: 'spelllang', 'spellfile', 'spellcapcheck' and 'spelloptions' which can all be local to the buffer. Consider calling this with nvim_buf_call().

Example:

```
vim.spell.check("the quik brown fox")
-- =>
-- {
--     {'quik', 'bad', 5}
-- }
```

Parameters:
- {str} (string)

Return:
([string, 'bad'|'rare'|'local'|'caps', integer][]) List of tuples with three items:
- The badly spelled word.
- The type of the spelling error: "bad" spelling mistake "rare" rare word "local" word only valid in another region "caps" word should start with Capital
- The position in {str} where the word begins.

Lua module: vim.system lua-vim-system

vim.SystemCompleted

Fields:
- {code} (integer)
- {signal} (integer)
- {stderr}? (string) nil if stderr is disabled or has a custom handler.
- {stdout}? (string) nil if stdout is disabled or has a custom handler.

vim.SystemObj

Fields:
- {cmd} (string[]) Command name and args
- {is_closing} (fun(self: vim.SystemObj): boolean) See SystemObj:is_closing().
- {kill} (fun(self: vim.SystemObj, signal: integer|string)) See SystemObj:kill().
- {pid} (integer) Process ID
- {wait} (fun(self: vim.SystemObj, timeout: integer?): vim.SystemCompleted) See SystemObj:wait().
- {write} (fun(self: vim.SystemObj, data: string[]|string?)) See SystemObj:write().

SystemObj:is_closing() SystemObj:is_closing()
Checks if the process handle is closing or already closed.

This method returns true if the underlying process handle is either nil or is in the process of closing. It is useful for determining whether it is safe to perform operations on the process handle.

Return:
(boolean)

SystemObj:kill({signal}) SystemObj:kill()
Sends a signal to the process.

The signal can be specified as an integer or as a string.

Example:

```
local obj = vim.system({'sleep', '10'})
obj:kill('sigterm') -- sends SIGTERM to the process
```

Parameters:
- {signal} (integer|string) Signal to send to the process. See luv-constants.

SystemObj:wait({timeout}) SystemObj:wait()
Waits for the process to complete or until the specified timeout elapses.

This method blocks execution until the associated process has exited or the optional timeout (in milliseconds) has been reached. If the process does not exit before the timeout, it is forcefully terminated with SIGKILL (signal 9), and the exit code is set to 124.

If no timeout is provided, the method will wait indefinitely (or use the timeout specified in the options when the process was started).

Example:

```
local obj = vim.system({'echo', 'hello'}, { text = true })
local result = obj:wait(1000) -- waits up to 1000ms
print(result.code, result.signal, result.stdout, result.stderr)
```

Parameters:
- {timeout} (integer?)

Return:
(vim.SystemCompleted)

SystemObj:write({data}) SystemObj:write()
Writes data to the stdin of the process or closes stdin.

If data is a list of strings, each string is written followed by a newline.

If data is a string, it is written as-is.

If data is nil, the write side of the stream is shut down and the pipe is closed.

Example:

```
local obj = vim.system({'cat'}, { stdin = true })
obj:write({'hello', 'world'}) -- writes 'hello\nworld\n' to stdin
obj:write(nil) -- closes stdin
```

Parameters:
- {data} (string[]|string?)

vim.system({cmd}, {opts}, {on_exit}) vim.system()
Runs a system command or throws an error if {cmd} cannot be run.

The command runs directly (not in 'shell') so shell builtins such as "echo" in cmd.exe, cmdlets in powershell, or "help" in bash, will not work unless you actually invoke a shell: vim.system({'bash', '-c', 'help'}).

Examples:

```
local on_exit = function(obj)
  print(obj.code)
  print(obj.signal)
  print(obj.stdout)
  print(obj.stderr)
end

-- Runs asynchronously:
vim.system({'echo', 'hello'}, { text = true }, on_exit)

-- Runs synchronously:
local obj = vim.system({'echo', 'hello'}, { text = true }):wait()
-- { code = 0, signal = 0, stdout = 'hello\n', stderr = '' }
```

See uv.spawn() for more details. **Note:** unlike uv.spawn(), vim.system throws an error if {cmd} cannot be run.

Parameters:
- {cmd} (string[]) Command to execute
- {opts} (table?) A table with the following fields:
  - {clear_env}? (boolean) env defines the job environment exactly, instead of merging current environment. **Note:** if env is nil, the current environment is used but without NVIM set.
  - {cwd}? (string) Set the current working directory for the sub-process.
  - {detach}? (boolean) Spawn the child process in a detached state - this will make it a process group leader, and will effectively enable the child to keep running after the parent exits. Note that the child process will still keep the parent's event loop alive unless the parent process calls uv.unref() on the child's process handle.
  - {env}? (table<string,string|number>) Set environment variables for the new process. Inherits the current environment with NVIM set to v:servername.
  - {stderr}? (fun(err:string?, data: string?)|boolean, default: true) Handle output from stderr.
  - {stdin}? (string|string[]|true) If true, then a pipe to stdin is opened and can be written to via the write() method to SystemObj. If string or string[] then will be written to stdin and closed.
  - {stdout}? (fun(err:string?, data: string?)|boolean, default: true) Handle output from stdout.
  - {text}? (boolean) Handle stdout and stderr as text. Normalizes line endings by replacing \r\n with \n.

- {timeout}? (integer) Run the command with a time limit in ms. Upon timeout the process is sent the TERM signal (15) and the exit code is set to 124.
- {on_exit} (fun(out: vim.SystemCompleted)?) Called when subprocess exits. When provided, the command runs asynchronously. See return of SystemObj:wait().

Overloads:
- fun(cmd: string[], on_exit: fun(out: vim.SystemCompleted)): vim.SystemObj

Return:
(vim.SystemObj)

Lua module: vim.text

vim.text

vim.text.diff({a}, {b}, {opts}) vim.text.diff()
Run diff on strings {a} and {b}. Any indices returned by this function, either directly or via callback arguments, are 1-based.

Examples:

```
vim.text.diff('a\n', 'b\nc\n')
-- =>
-- @@ -1 +1,2 @@
-- -a
-- +b
-- +c

vim.text.diff('a\n', 'b\nc\n', {result_type = 'indices'})
-- =>
-- {
--   {1, 1, 1, 2}
-- }
```

Parameters:
- {a} (string) First string to compare
- {b} (string) Second string to compare
- {opts} (table?) Optional parameters:
  - {algorithm}? ('myers'|'minimal'|'patience'|'histogram', default: 'myers') Diff algorithm to use. Values:
    - myers: the default algorithm
    - minimal: spend extra time to generate the smallest possible diff
    - patience: patience diff algorithm
    - histogram: histogram diff algorithm
  - {ctxlen}? (integer) Context length
  - {ignore_blank_lines}? (boolean) Ignore blank lines
  - {ignore_cr_at_eol}? (boolean) Ignore carriage return at end-of-line
  - {ignore_whitespace}? (boolean) Ignore whitespace
  - {ignore_whitespace_change}? (boolean) Ignore whitespace change
  - {ignore_whitespace_change_at_eol}? (boolean) Ignore whitespace change at end-of-line.
  - {indent_heuristic}? (boolean) Use the indent heuristic for the internal diff library.
  - {interhunkctxlen}? (integer) Inter hunk context length
  - {linematch}? (boolean|integer) Run linematch on the resulting hunks from xdiff. When integer, only hunks upto this size in lines are run through linematch. Requires result_type = indices, ignored otherwise.
  - {on_hunk}? (fun(start_a: integer, count_a: integer, start_b: integer, count_b: integer): integer?) Invoked for each hunk in the diff. Return a negative number to cancel the callback for any remaining hunks. Arguments:
    - start_a (integer): Start line of hunk in {a}.
    - count_a (integer): Hunk size in {a}.
    - start_b (integer): Start line of hunk in {b}.
    - count_b (integer): Hunk size in {b}.
  - {result_type}? ('unified'|'indices', default: 'unified') Form of the returned diff:
    - unified: String in unified format.
    - indices: Array of hunk locations. **Note:** This option is ignored if on_hunk is used.

Return:
(string|integer[][]?) See {opts.result_type}. nil if {opts.on_hunk} is given.

vim.text.hexdecode({enc}) vim.text.hexdecode()
Hex decode a string.

Parameters:
- {enc} (string) String to decode

Return (multiple):
(string?) Decoded string (string?) Error message, if any

vim.text.hexencode({str}) vim.text.hexencode()
Hex encode a string.

Parameters:
- {str} (string) String to encode

Return:
(string) Hex encoded string

vim.text.indent({size}, {text}, {opts}) vim.text.indent()
Sets the indent (i.e. the common leading whitespace) of non-empty lines in text to size spaces/tabs.

Indent is calculated by number of consecutive indent chars.

- The first indented, non-empty line decides the indent char (space/tab):
  - `SPC SPC TAB …` = two-space indent.
  - `TAB SPC …` = one-tab indent.
- Set `opts.expandtab` to treat tabs as spaces.

To "dedent" (remove the common indent), pass `size=0`:

```
vim.print(vim.text.indent(0, ' a\n  b\n'))
```

To adjust relative-to an existing indent, call indent() twice:

```
local indented, old_indent = vim.text.indent(0, ' a\n b\n')
indented = vim.text.indent(old_indent + 2, indented)
vim.print(indented)
```

To ignore the final, blank line when calculating the indent, use gsub() before calling indent():

```
local text = '  a\n  b\n '
vim.print(vim.text.indent(0, (text:gsub('\n[\t ]+\n?$', '\n'))))
```

Parameters:
- `{size}` (`integer`) Number of spaces.
- `{text}` (`string`) Text to indent.
- `{opts}` (`{ expandtab?: integer }?`)

Return (multiple):
(`string`) Indented text. (`integer`) Indent size before modification.

Lua module: vim.ui

vim.ui

vim.ui.input({opts}, {on_confirm}) vim.ui.input()

Prompts the user for input, allowing arbitrary (potentially asynchronous) work until `on_confirm`.

Example:

```
local opts = { prompt = 'Enter value for shiftwidth: ', scope = 'buffer' }
vim.ui.input(opts, function(input)
    vim.o.shiftwidth = tonumber(input)
end)
```

Parameters:
- `{opts}` (`table?`) Additional options. See input()
  - `{completion}?` (`string`) Specifies type of completion supported for input. Supported types are the same that can be supplied to a user-defined command using the "-complete=" argument. See :command-completion
  - `{default}?` (`string`) Default reply to the input
  - `{highlight}?` (`function`) Function that will be used for highlighting user inputs.
  - `{prompt}?` (`string`) Text of the prompt
  - `{scope}?` (`'cursor'|'line'|'buffer'|'window'|'tabpage'|'editor'|'project'`) Input scope, as in "This input is for something at cursor/line/etc scope". Can be used by `vim.ui.input` implementations to tweak behavior and presentation. For example, the input may adjust the floating window position: near the cursor if `cursor`, in window corner if `buffer` or `window`, etc.
- `{on_confirm}` (`fun(input?: string)`) Called once the user confirms or abort the input. `input` is what the user typed (it might be an empty string if nothing was entered), or `nil` if the user aborted the dialog.

vim.ui.open({path}, {opt}) vim.ui.open()

Opens `path` with the system default handler (macOS `open`, Windows `explorer.exe`, Linux `xdg-open`, …), or returns (but does not show) an error message on failure.

Can also be invoked with `:Open`. :Open

Expands "~/" and environment variables in filesystem paths.

Examples:

```
-- Asynchronous.
vim.ui.open("https://neovim.io/")
vim.ui.open("~/path/to/file")
-- Use the "osurl" command to handle the path or URL.
vim.ui.open("gh#neovim/neovim!29490", { cmd = { 'osurl' } })
-- Synchronous (wait until the process exits).
local cmd, err = vim.ui.open("$VIMRUNTIME")
if cmd then
  cmd:wait()
end
```

Parameters:
- `{path}` (`string`) Path or URL to open
- `{opt}` (`table?`) Options
  - `{cmd}?` (`string[]`) Command used to open the path or URL.

Return (multiple):
(`vim.SystemObj?`) Command object, or nil if not found. (`string?`) Error message on failure, or nil on success.

See also:
- vim.system()

vim.ui.progress_status() vim.ui.progress_status()
Gets a status description summarizing currently running progress messages. Convenient for inclusion in 'statusline'.

Return:
(`string`) Progress status

vim.ui.select({`items`}, {`opts`}, {`on_choice`}) vim.ui.select()
Prompts the user to pick from a list of items, allowing arbitrary (potentially asynchronous) work until `on_choice`. This is the standard "picker" interface, used by z=, :tselect, etc.

Plugins may override `vim.ui.select` to provide a custom picker; they are expected to call the `format_item` and `preview_item` handlers (if any) provided by the caller. They may also use the `kind` hint (if provided by the caller) to decide how to handle some items.

Note: the default `vim.ui.select` currently doesn't support preview.

Example:

```
vim.ui.select({ 'tabs', 'spaces' }, {
  prompt = 'Select tabs or spaces:',
  format_item = function(item)
    return ('I choose %s!'):format(item)
  end,
  preview_item = function(item)
    local lines = { 'This is ' .. vim.inspect(item) }
    local buf = vim.api.nvim_create_buf(false, true)
    vim.api.nvim_buf_set_lines(buf, 0, -1, false, lines)
    vim.bo[buf].bufhidden = 'wipe'
    return { buf = buf }
  end,
}, function(choice)
  vim.o.expandtab = choice == 'spaces'
  vim.print(('Selected "%s" => expandtab=%s'):format(choice, vim.o.expandtab))
end)
```

Parameters:
- {`items`} (`any[]`) Arbitrary items
- {`opts`} (`table`) Options
  - {`format_item`}? (`fun(item: any):string`, default: `tostring`) Decides how to format items when displayed in the picker.
  - {`kind`}? (`string`) Arbitrary hint string indicating the item shape. The picker may wish to use this to infer the structure or semantics of `items`, or the context in which select() was called.
  - {`preview_item`}? (`fun(item: any):{buf?:integer, pos?:[integer,integer], pos_end?:[integer,integer]}`) Decides how to preview an item by preparing a scratch buffer with the item details (including text, highlighting, etc.). When the picker decides to "preview" an item, it should call this function. Must return a table with these keys:
    - {`buf`}? (`integer`) Buffer containing the previewed item details, or nil if no preview should be shown.
    - {`pos`}? (`[integer, integer]`) Specifies the (1,0)-indexed cursor position in the preview buffer. If nil, should be treated as `{ 1, 0 }`.
    - {`pos_end`}? (`[integer, integer]`) Specifies the (1,0)-indexed (end-exclusive) end position of the preview range. If nil, no range is intended for a preview.
  - {`prompt`}? (`string`, default: `"Select one of:"`) Prompt text.
- {`on_choice`} (`fun(item: T?, idx: integer?)`) Called once the user made a choice. `idx` is the 1-based index of `item` within `items`, or `nil` if the user aborted the dialog.

Lua module: vim.ui.img vim.ui.img

EXPERIMENTAL: This API may change in the future. Its semantics are not yet finalized.

This provides a functional API for displaying images in Nvim. Currently supports PNG images via the Kitty graphics protocol.

To override the image backend, replace `vim.ui.img` with your own implementation providing set/get/del.

Examples:

```
-- Load image bytes from disk and display at row 5, column 10
local id = vim.ui.img.set(
 vim.fn.readblob('/path/to/img.png'),
 { row = 5, col = 10, width = 40, height = 20, zindex = 50 }
)

-- Update the image position
vim.ui.img.set(id, { row = 8, col = 12 })

-- Retrieve the current image opts
local opts = vim.ui.img.get(id)

-- Remove the image
vim.ui.img.del(id)

-- Remove all images
vim.ui.img.del(math.huge)
```

vim.ui.img.del({`id`}) vim.ui.img.del()
Delete an image, or all images if `math.huge` is given as the id.

Parameters:
- {`id`} (`integer`) image id, or `math.huge` to delete all images

Return:
(`boolean`) found true if any image existed

vim.ui.img.get({id}) vim.ui.img.get()
Get the opts for an image.

Parameters:
- {id} (integer)

Return:
(table?) opts copy of image opts, or nil if not found
- {col}? (integer) starting column (1-indexed)
- {height}? (integer) height in cells
- {row}? (integer) starting row (1-indexed)
- {width}? (integer) width in cells
- {zindex}? (integer) stacking order (higher = on top)

vim.ui.img.set({data_or_id}, {opts}) vim.ui.img.set()
Display an image or update an existing one.

When {data_or_id} is a string, displays the image bytes at the position given by {opts}. Returns an integer id for later use.

When {data_or_id} is an integer (a previously returned id), updates the image with new {opts}.

Parameters:
- {data_or_id} (string|integer) image bytes (string) or existing id (integer)
- {opts} (table?) A table with the following fields:
  - {col}? (integer) starting column (1-indexed)
  - {height}? (integer) height in cells
  - {row}? (integer) starting row (1-indexed)
  - {width}? (integer) width in cells
  - {zindex}? (integer) stacking order (higher = on top)

Return:
(integer) id

Lua module: vim.uri vim.uri

vim.uri_decode({str}) vim.uri_decode()
URI-decodes a string containing percent escapes.

Parameters:
- {str} (string) string to decode

Return:
(string) decoded string

vim.uri_encode({str}, {rfc}) vim.uri_encode()
URI-encodes a string using percent escapes.

Parameters:
- {str} (string) string to encode
- {rfc} ("rfc2396"|"rfc2732"|"rfc3986"?)

Return:
(string) encoded string

vim.uri_from_bufnr({buf}) vim.uri_from_bufnr()
Gets a URI from a bufnr.

Parameters:
- {buf} (integer)

Return:
(string) URI

vim.uri_from_fname({path}) vim.uri_from_fname()
Gets a URI from a file path.

Parameters:
- {path} (string) Path to file

Return:
(string) URI

vim.uri_to_bufnr({uri}) vim.uri_to_bufnr()
Gets the buffer for a uri. Creates a new unloaded buffer if no buffer for the uri already exists.

Parameters:
- {uri} (string)

Return:
(integer) bufnr

vim.uri_to_fname({uri}) vim.uri_to_fname()
Gets a filename from a URI.

Parameters:
- {uri} (string)

Return:
(string) filename or unchanged URI for non-file URIs

Lua module: vim.version

vim.version

The `vim.version` module provides functions for comparing versions and ranges conforming to the https://semver.org spec. Plugins, and plugin managers, can use this to check available tools and dependencies on the current system.

Example:

```
local v = vim.version.parse(vim.system({'tmux', '-V'}):wait().stdout, {strict=false})
if vim.version.gt(v, {3, 2, 0}) then
  -- ...
end
```

vim.version() returns the version of the current Nvim process.

VERSION RANGE SPEC

version-range

A version "range spec" defines a semantic version range which can be tested against a version, using vim.version.range().

Supported range specs are shown in the following table. **Note:** suffixed versions (1.2.3-rc1) are not matched.

```
1.2.3            is 1.2.3
=1.2.3           is 1.2.3
>1.2.3           greater than 1.2.3
<1.2.3           before 1.2.3
>=1.2.3          at least 1.2.3
<=1.2.3          at most 1.2.3
~1.2.3           is >=1.2.3 <1.3.0       "reasonably close to 1.2.3"
^1.2.3           is >=1.2.3 <2.0.0       "compatible with 1.2.3"
^0.2.3           is >=0.2.3 <0.3.0       (0.x.x is special)
^0.0.1           is =0.0.1               (0.0.x is special)
^1.2             is >=1.2.0 <2.0.0       (like ^1.2.0)
~1.2             is >=1.2.0 <1.3.0       (like ~1.2.0)
^1               is >=1.0.0 <2.0.0       "compatible with 1"
~1               same                    "reasonably close to 1"
1.x              same
1.*              same
1                same
*                any version
x                same

1.2.3 - 2.3.4    is >=1.2.3 <2.3.4

Partial right: missing pieces treated as x (2.3 => 2.3.x).
1.2.3 - 2.3      is >=1.2.3 <2.4.0
1.2.3 - 2        is >=1.2.3 <3.0.0

Partial left: missing pieces treated as 0 (1.2 => 1.2.0).
1.2 - 2.3.0      is 1.2.0 - 2.3.0
```

vim.VersionRange

Fields:

- {from} (vim.Version)
- {has} (fun(self: vim.VersionRange, version: string|vim.Version): boolean) See VersionRange:has().
- {to}? (vim.Version)

VersionRange:has({version})

VersionRange:has()

Check if a version is in the range (inclusive `from`, exclusive `to`).

Example:

```
local r = vim.version.range('1.0.0 - 2.0.0')
print(r:has('1.9.9'))       -- true
print(r:has('2.0.0'))       -- false
print(r:has(vim.version())) -- check against current Nvim version
```

Or use cmp(), le(), lt(), ge(), gt(), and/or eq() to compare a version against `.to` and `.from` directly:

```
local r = vim.version.range('1.0.0 - 2.0.0') -- >=1.0, <2.0
print(vim.version.ge({1,0,3}, r.from) and vim.version.lt({1,0,3}, r.to))
```

Attributes:
Since: 0.9.0

Parameters:

- {version} (string|vim.Version)

Return:
(boolean)

See also:

- https://github.com/npm/node-semver#ranges

vim.version()

vim.version()

Returns the current Nvim version.

Return:
(vim.Version)

vim.version.cmp({v1}, {v2}) vim.version.cmp()
Parses and compares two version objects (the result of vim.version.parse(), or specified literally as a `{major, minor, patch}` tuple, e.g. `{1, 0, 3}`).

Example:

```
if vim.version.cmp({1,0,3}, {0,2,1}) == 0 then
  -- ...
end
local v1 = vim.version.parse('1.0.3-pre')
local v2 = vim.version.parse('0.2.1')
if vim.version.cmp(v1, v2) == 0 then
  -- ...
end
```

Note:
- Per semver, build metadata is ignored when comparing two otherwise-equivalent versions.

Attributes:
Since: 0.9.0

Parameters:
- `{v1}` (`vim.Version|number[]|string`) Version object.
- `{v2}` (`vim.Version|number[]|string`) Version to compare with v1.

Return:
(`integer`) -1 if `v1 < v2`, 0 if `v1 == v2`, 1 if `v1 > v2`.

vim.version.eq({v1}, {v2}) vim.version.eq()
Returns `true` if the given versions are equal. See vim.version.cmp() for usage.

Attributes:
Since: 0.9.0

Parameters:
- `{v1}` (`vim.Version|number[]|string`)
- `{v2}` (`vim.Version|number[]|string`)

Return:
(`boolean`)

vim.version.ge({v1}, {v2}) vim.version.ge()
Returns `true` if `v1 >= v2`. See vim.version.cmp() for usage.

Attributes:
Since: 0.10.0

Parameters:
- `{v1}` (`vim.Version|number[]|string`)
- `{v2}` (`vim.Version|number[]|string`)

Return:
(`boolean`)

vim.version.gt({v1}, {v2}) vim.version.gt()
Returns `true` if `v1 > v2`. See vim.version.cmp() for usage.

Attributes:
Since: 0.9.0

Parameters:
- `{v1}` (`vim.Version|number[]|string`)
- `{v2}` (`vim.Version|number[]|string`)

Return:
(`boolean`)

vim.version.intersect({r1}, {r2}) vim.version.intersect()
Computes the common range shared by the given ranges.

Attributes:
Since: 0.12.0

Parameters:
- `{r1}` (`vim.VersionRange`) First range to intersect.
- `{r2}` (`vim.VersionRange`) Second range to intersect.

Return:
(`vim.VersionRange?`) Maximal range that is present inside both `r1` and `r2`. `nil` if such range does not exist.

vim.version.last({versions}) vim.version.last()
TODO: generalize this, move to func.lua

Parameters:
- `{versions}` (`vim.Version[]`)

Return:
(`vim.Version?`)

vim.version.le({v1}, {v2}) vim.version.le()
Returns `true` if `v1 <= v2`. See vim.version.cmp() for usage.

Attributes:
Since: 0.10.0

Parameters:
- {v1} (vim.Version|number[]|string)
- {v2} (vim.Version|number[]|string)

Return:
(boolean)

vim.version.lt({v1}, {v2}) vim.version.lt()
Returns true if v1 < v2. See vim.version.cmp() for usage.

Attributes:
Since: 0.9.0

Parameters:
- {v1} (vim.Version|number[]|string)
- {v2} (vim.Version|number[]|string)

Return:
(boolean)

vim.version.parse({version}, {opts}) vim.version.parse()
Parses a semantic version string and returns a version object which can be used with other vim.version functions. For example "1.0.1-rc1+build.2" returns:

```
{ major = 1, minor = 0, patch = 1, prerelease = "rc1", build = "build.2" }
```

Attributes:
Since: 0.9.0

Parameters:
- {version} (string) Version string to parse.
- {opts} (table?) Options for parsing.
  - {strict}? (boolean, default: false) If true, no coercion is attempted on input not conforming to semver v2.0.0. If false, parse() attempts to coerce input such as "1.0", "0-x", "tmux 3.2a" into valid versions.

Return:
(vim.Version?) Version object or nil if input is invalid.

See also:
- https://semver.org/spec/v2.0.0.html

vim.version.range({spec}) vim.version.range()
Parses a semver version-range "spec" and returns vim.VersionRange object:

Attributes:
Since: 0.9.0

Parameters:
- {spec} (string) Version range "spec"

Return:
(vim.VersionRange?)

UI2 ui2

WARNING: This is an experimental feature intended to replace the builtin message + cmdline presentation layer.

To enable this feature (default opts shown, see also 'messagesopt'):

```
require('vim._core.ui2').enable({
  enable = true, -- Whether to enable or disable the UI.
  msg = { -- Options related to the message module.
    ---@type string|table<string, 'cmd'|'msg'|'pager'> Default message target
    ---or table mapping |ui-messages| kinds, triggers and IDs to a target.
    ---Table keys are matched as a Lua pattern to the message ID. 'default'
    ---mapping applies to any omitted kind: { default = 'cmd', progress = 'msg' }.
    targets = 'cmd',
    dialog = { -- Options related to dialog window.
      height = 0.5, -- Maximum height.
    },
    msg = { -- Options related to msg window.
      height = 0.5, -- Maximum height.
    },
    pager = { -- Options related to message window.
      height = 0.999, -- Maximum height.
    },
  },
})
```

ui2 behavior is also configured by these 'messagesopt' items: "pager", "timeout". Example:

```
set messagesopt+=maxheight:50,pager:<CR>,timeout:4000
```

There are four special windows/buffers for presenting messages and cmdline:
- "cmd": Cmdline. Also used by default for messages, 'showcmd', 'showmode', 'ruler'.
- "msg": Message window. Shows ephemeral messages; useful if 'cmdheight' is 0.

- "pager": Pager window. Shows :messages, allows navigating all messages.
- "dialog": Dialog window. Shows modal prompts that wait for user input.

In each of the special buffers, 'filetype' is set as listed above ("cmd", "msg", ...). Handle the FileType event to configure any local options for these windows/buffers.

Unlike the legacy hit-enter prompt, messages exceeding 'cmdheight' are collapsed, followed by a `[+x]` "spill" indicator, where x is number of overflow lines. To see all messages, do either:
- g< at any time.
- Type the "pager" char (if configured in 'messagesopt'), immediately after an interactive : command.

LUA REFERENCE MANUAL

Version 0.3.0 August 7th, 2022

Vimdoc version (c) 2006 by Luis Carvalho <lexcarvalho at gmail dot com>

Adapted from "Lua: 5.1 reference manual" R. Ierusalimschy, L. H. de Figueiredo, W. Celes Copyright (c) 2006 Lua.org, PUC-Rio.

See lua-ref-doc for information on this manual. See lua-ref-copyright for copyright and licenses.

1 INTRODUCTION luaref-intro

Lua is an extension programming language designed to support general procedural programming with data description facilities. It also offers good support for object-oriented programming, functional programming, and data-driven programming. Lua is intended to be used as a powerful, light-weight scripting language for any program that needs one. Lua is implemented as a library, written in clean C (that is, in the common subset of ANSI C and C++).

Being an extension language, Lua has no notion of a "main" program: it only works embedded in a host client, called the embedding program or simply the host. This host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code. Through the use of C functions, Lua can be augmented to cope with a wide range of different domains, thus creating customized programming languages sharing a syntactical framework.

Lua is free software, and is provided as usual with no guarantees, as stated in its license. The implementation described in this manual is available at Lua's official web site, www.lua.org.

Like any other reference manual, this document is dry in places. For a discussion of the decisions behind the design of Lua, see references at lua-ref-bibliography. For a detailed introduction to programming in Lua, see Roberto's book, Programming in Lua.

Lua means "moon" in Portuguese and is pronounced LOO-ah.

2 THE LANGUAGE lua-language

This section describes the lexis, the syntax, and the semantics of Lua. In other words, this section describes which tokens are valid, how they can be combined, and what their combinations mean.

The language constructs will be explained using the usual extended BNF notation, in which `{ a }` means 0 or more a's, and `[ a ]` means an optional a.

2.1 Lexical Conventions lua-lexical

lua-names lua-identifiers Names (also called identifiers) in Lua can be any string of letters, digits, and underscores, not beginning with a digit. This coincides with the definition of identifiers in most languages. (The definition of letter depends on the current locale: any character considered alphabetic by the current locale can be used in an identifier.) Identifiers are used to name variables and table fields.

The following keywords are reserved and cannot be used as names:

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
repeat    return    then      true      until     while
```

Lua is a case-sensitive language: `and` is a reserved word, but `And` and `AND` are two different, valid names. As a convention, names starting with an underscore followed by uppercase letters (such as `_VERSION`) are reserved for internal global variables used by Lua.

The following strings denote other tokens:

```
+     -     *     /     %     ^     #
==    ~=    <=    >=    <     >     =
(     )     {     }     [     ]
;     :     ,     .     ..    ...
```

lua-literal

Literal strings can be delimited by matching single or double quotes, and can contain the following C-like escape sequences:

- `\a` bell
- `\b` backspace
- `\f` form feed
- `\n` newline
- `\r` carriage return
- `\t` horizontal tab
- `\v` vertical tab
- `\\` backslash
- `\"` quotation mark (double quote)
- `\'` apostrophe (single quote)

Moreover, a backslash followed by a real newline results in a newline in the string. A character in a string may also be specified by its numerical value using the escape sequence `\ddd`, where ddd is a sequence of up to three decimal digits. (Note that if a numerical escape is to be followed by a digit, it must be expressed using exactly three digits.) Strings in Lua may contain any 8-bit value, including embedded zeros, which can be specified as `\0`.

To put a double (single) quote, a newline, a backslash, or an embedded zero inside a literal string enclosed by double (single) quotes you must use an escape sequence. Any other character may be directly inserted into the literal. (Some control characters may cause problems for the file system, but Lua has no problem with them.)

Literal strings can also be defined using a long format enclosed by long brackets. We define an opening long bracket of level n as an opening square bracket followed by n equal signs followed by another opening square bracket. So, an opening long bracket of level 0 is written as `[[`,

an opening long bracket of level 1 is written as `[=[`, and so on. A closing long bracket is defined similarly; for instance, a closing long bracket of level 4 is written as `]====]`. A long string starts with an opening long bracket of any level and ends at the first closing long bracket of the same level. Literals in this bracketed form may run for several lines, do not interpret any escape sequences, and ignore long brackets of any other level. They may contain anything except a closing bracket of the proper level.

For convenience, when the opening long bracket is immediately followed by a newline, the newline is not included in the string. As an example, in a system using ASCII (in which a is coded as 97, newline is coded as 10, and 1 is coded as 49), the five literals below denote the same string:

```
a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
a = [==[
alo
123"]==]
```

lua-numconstant

A numerical constant may be written with an optional decimal part and an optional decimal exponent. Lua also accepts integer hexadecimal constants, by prefixing them with `0x`. Examples of valid numerical constants are

```
3     3.0     3.1416  314.16e-2   0.31416E1   0xff   0x56
```

lua-comment

A comment starts with a double hyphen (`--`) anywhere outside a string. If the text immediately after `--` is not an opening long bracket, the comment is a short comment, which runs until the end of the line. Otherwise, it is a long comment, which runs until the corresponding closing long bracket. Long comments are frequently used to disable code temporarily.

2.2 Values and Types

lua-values

Lua is a dynamically typed language. This means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

All values in Lua are first-class values. This means that all values can be stored in variables, passed as arguments to other functions, and returned as results.

lua-types lua-nil lua-true lua-false lua-number lua-string There are eight basic types in Lua: `nil`, `boolean`, `number`, `string`, `function`, `userdata`, `thread`, and `table`. Nil is the type of the value `nil`, whose main property is to be different from any other value; it usually represents the absence of a useful value. Boolean is the type of the values `false` and `true`. Both `nil` and `false` make a condition false; any other value makes it true. Number represents real (double-precision floating-point) numbers. (It is easy to build Lua interpreters that use other internal representations for numbers, such as single-precision float or long integers; see file `luaconf.h`.) String represents arrays of characters. Lua is 8-bit clean: strings may contain any 8-bit character, including embedded zeros (`\0`) (see lua-literal).

Lua can call (and manipulate) functions written in Lua and functions written in C (see lua-function).

lua-userdatatype

The type userdata is provided to allow arbitrary C data to be stored in Lua variables. This type corresponds to a block of raw memory and has no pre-defined operations in Lua, except assignment and identity test. However, by using metatables, the programmer can define operations for userdata values (see lua-metatable). Userdata values cannot be created or modified in Lua, only through the C API. This guarantees the integrity of data owned by the host program.

lua-thread

The type `thread` represents independent threads of execution and it is used to implement coroutines (see lua-coroutine). Do not confuse Lua threads with operating-system threads. Lua supports coroutines on all systems, even those that do not support threads.

lua-table

The type `table` implements associative arrays, that is, arrays that can be indexed not only with numbers, but with any value (except `nil`). Tables can be heterogeneous; that is, they can contain values of all types (except `nil`). Tables are the sole data structuring mechanism in Lua; they may be used to represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc. To represent records, Lua uses the field name as an index. The language supports this representation by providing `a.name` as syntactic sugar for `a["name"]`. There are several convenient ways to create tables in Lua (see lua-tableconstructor).

Like indices, the value of a table field can be of any type (except `nil`). In particular, because functions are first-class values, table fields may contain functions. Thus tables may also carry methods (see lua-function-define).

Tables, functions, threads and (full) userdata values are objects: variables do not actually contain these values, only references to them. Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy.

The library function `type` returns a string describing the type of a given value (see lua-type()).

2.2.1 Coercion

lua-coercion

Lua provides automatic conversion between string and number values at run time. Any arithmetic operation applied to a string tries to convert that string to a number, following the usual conversion rules. Conversely, whenever a number is used where a string is expected, the number is converted to a string, in a reasonable format. For complete control of how numbers are converted to strings, use the `format` function from the string library (see string.format()).

2.3 Variables

lua-variables

Variables are places that store values. There are three kinds of variables in Lua: global variables, local variables, and table fields.

A single name can denote a global variable or a local variable (or a function's formal parameter, which is a particular form of local variable):

```
var ::= Name
```

Name denotes identifiers, as defined in lua-lexical.

Any variable is assumed to be global unless explicitly declared as a local (see lua-local). Local variables are lexically scoped: local variables can be freely accessed by functions defined inside their scope (see lua-visibility).

Before the first assignment to a variable, its value is `nil`.

Square brackets are used to index a table:

```
var ::= prefixexp [ exp ]
```

The first expression (`prefixexp`) should result in a table value; the second expression (`exp`) identifies a specific entry inside that table. The expression denoting the table to be indexed has a restricted syntax; see lua-expressions for details.

The syntax `var.NAME` is just syntactic sugar for `var["NAME"]` :

```
var ::= prefixexp . Name
```

All global variables live as fields in ordinary Lua tables, called environment tables or simply environments (see lua-environments). Each function has its own reference to an environment, so that all global variables in this function will refer to this environment table. When a function is created, it inherits the environment from the function that created it. To get the environment table of a Lua function, you call `getfenv` (see lua_getfenv()). To replace it, you call `setfenv` (see setfenv()). (You can only manipulate the environment of C functions through the debug library; see lua-lib-debug.)

An access to a global variable `x` is equivalent to `_env.x`, which in turn is equivalent to

```
gettable_event(_env, "x")
```

where _env is the environment of the running function. (The _env variable is not defined in Lua. We use it here only for explanatory purposes.)

The meaning of accesses to global variables and table fields can be changed via metatables. An access to an indexed variable `t[i]` is equivalent to a call `gettable_event(t,i)`. (See lua-metatable for a complete description of the `gettable_event` function. This function is not defined or callable in Lua. We use it here only for explanatory purposes.)

2.4 Statements **lua-statement**

Lua supports an almost conventional set of statements, similar to those in Pascal or C. This set includes assignment, control structures, function calls, and variable declarations.

2.4.1 Chunks **lua-chunk**

The unit of execution of Lua is called a chunk. A chunk is simply a sequence of statements, which are executed sequentially. Each statement can be optionally followed by a semicolon:

```
chunk ::= {stat [ ; ]}
```

There are no empty statements and thus `;;` is not legal.

Lua handles a chunk as the body of an anonymous function with a variable number of arguments (see lua-function-define). As such, chunks can define local variables, receive arguments, and return values.

A chunk may be stored in a file or in a string inside the host program. When a chunk is executed, first it is pre-compiled into instructions for a virtual machine, and then the compiled code is executed by an interpreter for the virtual machine.

Chunks may also be pre-compiled into binary form; see program `luac` for details. Programs in source and compiled forms are interchangeable; Lua automatically detects the file type and acts accordingly.

2.4.2 Blocks **lua-block**

A block is a list of statements; syntactically, a block is the same as a chunk:

```
block ::= chunk
```

lua-do lua-end A block may be explicitly delimited to produce a single statement:

```
stat ::= do block end
```

Explicit blocks are useful to control the scope of variable declarations. Explicit blocks are also sometimes used to add a `return` or `break` statement in the middle of another block (see lua-control).

2.4.3 Assignment **lua-assign**

Lua allows multiple assignment. Therefore, the syntax for assignment defines a list of variables on the left side and a list of expressions on the right side. The elements in both lists are separated by commas:

```
stat ::= varlist1 = explist1
varlist1 ::= var { , var }
explist1 ::= exp { , exp }
```

Expressions are discussed in lua-expressions.

Before the assignment, the list of values is adjusted to the length of the list of variables. If there are more values than needed, the excess values are thrown away. If there are fewer values than needed, the list is extended with as many `nils` as needed. If the list of expressions

ends with a function call, then all values returned by this call enter in the list of values, before the adjustment (except when the call is enclosed in parentheses; see lua-expressions).

The assignment statement first evaluates all its expressions and only then are the assignments performed. Thus the code

```
i = 3
i, a[i] = i+1, 20
```

sets a[3] to 20, without affecting a[4] because the i in a[i] is evaluated (to 3) before it is assigned 4. Similarly, the line

```
x, y = y, x
```

exchanges the values of x and y.

The meaning of assignments to global variables and table fields can be changed via metatables. An assignment to an indexed variable t[i] = val is equivalent to settable_event(t,i,val). (See lua-metatable for a complete description of the settable_event function. This function is not defined or callable in Lua. We use it here only for explanatory purposes.)

An assignment to a global variable x = val is equivalent to the assignment _env.x = val, which in turn is equivalent to

```
settable_event(_env, "x", val)
```

where _env is the environment of the running function. (The _env variable is not defined in Lua. We use it here only for explanatory purposes.)

2.4.4 Control Structures

lua-control

lua-if lua-then lua-else lua-elseif lua-while lua-repeat lua-until The control structures if, while, and repeat have the usual meaning and familiar syntax:

```
stat ::=  while  exp do block end
stat ::=  repeat  block until exp
stat ::=  if  exp then block { elseif exp then block }
          [ else block ] end
```

Lua also has a for statement, in two flavors (see lua-for).

The condition expression of a control structure may return any value. Both false and nil are considered false. All values different from nil and false are considered true (in particular, the number 0 and the empty string are also true).

In the repeat-until loop, the inner block does not end at the until keyword, but only after the condition. So, the condition can refer to local variables declared inside the loop block.

lua-return

The return statement is used to return values from a function or a chunk (which is just a function). Functions and chunks may return more than one value, so the syntax for the return statement is

stat ::= return [explist1]

lua-break

The break statement is used to terminate the execution of a while, repeat, or for loop, skipping to the next statement after the loop:

stat ::= break

A break ends the innermost enclosing loop.

The return and break statements can only be written as the last statement of a block. If it is really necessary to return or break in the middle of a block, then an explicit inner block can be used, as in the idioms do return end and do break end, because now return and break are the last statements in their (inner) blocks.

2.4.5 For Statement

for lua-for

The for statement has two forms: one numeric and one generic.

The numeric for loop repeats a block of code while a control variable runs through an arithmetic progression. It has the following syntax:

```
stat ::=  for  Name = exp , exp [ , exp ] do block end
```

The block is repeated for name starting at the value of the first exp, until it passes the second exp by steps of the third exp. More precisely, a for statement like

for var = e1, e2, e3 do block end

is equivalent to the code:

```
do
  local  var, limit, step  = tonumber(e1), tonumber(e2), tonumber(e3)
  if not (  var  and  limit  and  step  ) then error() end
  while (  step  >0 and  var  <=  limit  )
          or (  step  <=0 and  var  >=  limit  ) do
     block
     var  =  var  +  step
  end
end
```

Note the following:

- All three control expressions are evaluated only once, before the loop starts. They must all result in numbers.
- var, limit and step are invisible variables. The names are here for explanatory purposes only.
- If the third expression (the step) is absent, then a step of 1 is used.

- You can use `break` to exit a `for` loop.
- The loop variable `var` is local to the loop; you cannot use its value after the `for` ends or is broken. If you need this value, assign it to another variable before breaking or exiting the loop.

for-in

The generic `for` statement works over functions, called iterators. On each iteration, the iterator function is called to produce a new value, stopping when this new value is `nil`. The generic `for` loop has the following syntax:

```
stat ::=  for  namelist in explist1 do block end
namelist ::= Name { , Name }
```

A `for` statement like

`for var1, ..., varn in explist do block end`

is equivalent to the code:

```
do
  local  f, s, var  =  explist
  while true do
      local  var1, ..., varn  =  f(s, var)
      var  =  var1
      if  var  == nil then break end
      block
  end
end
```

Note the following:

- `explist` is evaluated only once. Its results are an iterator function, a `state`, and an initial value for the first iterator variable.
- `f`, `s`, and `var` are invisible variables. The names are here for explanatory purposes only.
- You can use `break` to exit a `for` loop.
- The loop variables `var1, ..., varn` are local to the loop; you cannot use their values after the `for` ends. If you need these values, then assign them to other variables before breaking or exiting the loop.

2.4.6 Function Calls as Statements

lua-funcstatement

To allow possible side-effects, function calls can be executed as statements:

```
stat ::= functioncall
```

In this case, all returned values are thrown away. Function calls are explained in lua-function.

2.4.7 Local Declarations

lua-local

Local variables may be declared anywhere inside a block. The declaration may include an initial assignment:

```
stat ::=  local  namelist [ = explist1 ]
namelist ::= Name { , Name }
```

If present, an initial assignment has the same semantics of a multiple assignment (see lua-assign). Otherwise, all variables are initialized with `nil`.

A chunk is also a block (see lua-chunk), and so local variables can be declared in a chunk outside any explicit block. The scope of such local variables extends until the end of the chunk.

The visibility rules for local variables are explained in lua-visibility.

2.5 Expressions

lua-expressions

The basic expressions in Lua are the following:

```
exp ::= prefixexp
exp ::=  nil  |  false  |  true
exp ::= Number
exp ::= String
exp ::= function
exp ::= tableconstructor
exp ::= ...
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var | functioncall | ( exp )
```

Numbers and literal strings are explained in lua-lexical; variables are explained in lua-variables; function definitions are explained in lua-function-define; function calls are explained in lua-function; table constructors are explained in lua-tableconstructor. Vararg expressions, denoted by three dots (`...`), can only be used inside vararg functions; they are explained in lua-function-define.

Binary operators comprise arithmetic operators (see lua-arithmetic), relational operators (see lua-relational), logical operators (see lua-logicalop), and the concatenation operator (see lua-concat). Unary operators comprise the unary minus (see lua-arithmetic), the unary `not` (see lua-logicalop), and the unary length operator (see lua-length).

Both function calls and vararg expressions may result in multiple values. If the expression is used as a statement (see lua-funcstatement) (only possible for function calls), then its return list is adjusted to zero elements, thus discarding all returned values. If the expression is used as the last (or the only) element of a list of expressions, then no adjustment is made (unless the call is enclosed in parentheses). In all other contexts, Lua adjusts the result list to one element, discarding all values except the first one.

Here are some examples:

```
f()                -- adjusted to 0 results
g(f(), x)          -- f() is adjusted to 1 result
g(x, f())          -- g gets x plus all results from f()
a,b,c = f(), x     -- f() is adjusted to 1 result (c gets nil)
a,b = ...          -- a gets the first vararg parameter, b gets
                   -- the second (both a and b may get nil if there
                   -- is no corresponding vararg parameter)

a,b,c = x, f()     -- f() is adjusted to 2 results
a,b,c = f()        -- f() is adjusted to 3 results
return f()         -- returns all results from f()
return ...         -- returns all received vararg parameters
return x,y,f()     -- returns x, y, and all results from f()
{f()}              -- creates a list with all results from f()
{...}              -- creates a list with all vararg parameters
{f(), nil}         -- f() is adjusted to 1 result
```

An expression enclosed in parentheses always results in only one value. Thus, `(f(x,y,z))` is always a single value, even if `f` returns several values. (The value of `(f(x,y,z))` is the first value returned by `f` or `nil` if `f` does not return any values.)

2.5.1 Arithmetic Operators

lua-arithmetic

Lua supports the usual arithmetic operators: the binary + (addition), - (subtraction), * (multiplication), / (division), % (modulo) and ^ (exponentiation); and unary - (negation). If the operands are numbers, or strings that can be converted to numbers (see lua-coercion), then all operations have the usual meaning. Exponentiation works for any exponent. For instance, `x^(-0.5)` computes the inverse of the square root of x. Modulo is defined as

```
a % b == a - math.floor(a/b)*b
```

That is, it is the remainder of a division that rounds the quotient towards minus infinity.

2.5.2 Relational Operators

lua-relational

The relational operators in Lua are

```
==    ~=    <     >     <=    >=
```

These operators always result in `false` or `true`.

Equality (==) first compares the type of its operands. If the types are different, then the result is `false`. Otherwise, the values of the operands are compared. Numbers and strings are compared in the usual way. Objects (tables, userdata, threads, and functions) are compared by reference: two objects are considered equal only if they are the same object. Every time you create a new object (a table, userdata, or function), this new object is different from any previously existing object.

You can change the way that Lua compares tables and userdata using the "eq" metamethod (see lua-metatable).

The conversion rules of coercion lua-coercion do not apply to equality comparisons. Thus, `"0"==0` evaluates to `false`, and `t[0]` and `t["0"]` denote different entries in a table.

The operator ~= is exactly the negation of equality (==).

The order operators work as follows. If both arguments are numbers, then they are compared as such. Otherwise, if both arguments are strings, then their values are compared according to the current locale. Otherwise, Lua tries to call the "lt" or the "le" metamethod (see lua-metatable).

2.5.3 Logical Operators

lua-logicalop

The logical operators in Lua are

```
and    or    not
```

Like the control structures (see lua-control), all logical operators consider both `false` and `nil` as false and anything else as true.

lua-not lua-and lua-or The negation operator `not` always returns `false` or `true`. The conjunction operator `and` returns its first argument if this value is `false` or `nil`; otherwise, and returns its second argument. The disjunction operator `or` returns its first argument if this value is different from `nil` and `false`; otherwise, `or` returns its second argument. Both `and` and `or` use short-cut evaluation, that is, the second operand is evaluated only if necessary. Here are some examples:

```
10 or 20            --> 10
10 or error()       --> 10
nil or "a"          --> "a"
nil and 10          --> nil
false and error()   --> false
false and nil       --> false
false or nil        --> nil
10 and 20           --> 20
```

(In this manual, `-->` indicates the result of the preceding expression.)

2.5.4 Concatenation

lua-concat

The string concatenation operator in Lua is denoted by two dots (`..`). If both operands are strings or numbers, then they are converted to strings according to the rules mentioned in lua-coercion. Otherwise, the "concat" metamethod is called (see lua-metatable).

2.5.5 The Length Operator

lua-# lua-length

The length operator is denoted by the unary operator #. The length of a string is its number of bytes (that is, the usual meaning of string length when each character is one byte).

The length of a table `t` is defined to be any integer index `n` such that `t[n]` is not `nil` and `t[n+1]` is `nil`; moreover, if `t[1]` is `nil`, `n` may be zero. For a regular array, with non-nil values from 1 to a given `n`, its length is exactly that `n`, the index of its last value. If the array has "holes" (that is, `nil` values between other non-nil values), then `#t` may be any of the indices that directly precedes a `nil` value (that is, it may consider any such `nil` value as the end of the array).

2.5.6 Precedence

lua-precedence

Operator precedence in Lua follows the table below, from lower to higher priority:

```
or
and
<     >     <=    >=    ~=    ==
..
+     -
*     /
not   #     - (unary)
^
```

As usual, you can use parentheses to change the precedences in an expression. The concatenation (`..`) and exponentiation (^) operators are right associative. All other binary operators are left associative.

2.5.7 Table Constructors

lua-tableconstructor

Table constructors are expressions that create tables. Every time a constructor is evaluated, a new table is created. Constructors can be used to create empty tables, or to create a table and initialize some of its fields. The general syntax for constructors is

```
tableconstructor ::= { [ fieldlist ] }
fieldlist ::= field { fieldsep field } [ fieldsep ]
field ::= [ exp ]  = exp | Name = exp | exp
fieldsep ::=  , |  ;
```

Each field of the form `[exp1] = exp2` adds to the new table an entry with key `exp1` and value `exp2`. A field of the form `name = exp` is equivalent to `["name"] = exp`. Finally, fields of the form `exp` are equivalent to `[i] = exp`, where `i` are consecutive numerical integers, starting with 1. Fields in the other formats do not affect this counting. For example,

```
a = { [f(1)] = g; "x", "y"; x = 1, f(x), [30] = 23; 45 }
```

is equivalent to

```
do
  local t = {}
  t[f(1)] = g
  t[1] = "x"         -- 1st exp
  t[2] = "y"         -- 2nd exp
  t.x = 1            -- temp["x"] = 1
  t[3] = f(x)        -- 3rd exp
  t[30] = 23
  t[4] = 45          -- 4th exp
  a = t
end
```

If the last field in the list has the form exp and the expression is a function call, then all values returned by the call enter the list consecutively (see lua-function). To avoid this, enclose the function call in parentheses (see lua-expressions).

The field list may have an optional trailing separator, as a convenience for machine-generated code.

2.5.8 Function Calls

lua-function

A function call in Lua has the following syntax:

```
functioncall ::= prefixexp args
```

In a function call, first `prefixexp` and `args` are evaluated. If the value of `prefixexp` has type `function`, then this function is called with the given arguments. Otherwise, the `prefixexp` "call" metamethod is called, having as first parameter the value of `prefixexp`, followed by the original call arguments (see lua-metatable).

The form

```
functioncall ::= prefixexp : Name args
```

can be used to call "methods". A call `v:name( args )` is syntactic sugar for `v.name(v, args )`, except that v is evaluated only once.

Arguments have the following syntax:

```
args ::=  ( [ explist1 ] )
args ::= tableconstructor
args ::= String
```

All argument expressions are evaluated before the call. A call of the form `f{ fields }` is syntactic sugar for `f({ fields })`, that is, the argument list is a single new table. A call of the form `f' string '` (or `f" string "` or `f[[ string ]]`) is syntactic sugar for `f(' string ')`, that is, the argument list is a single literal string.

As an exception to the free-format syntax of Lua, you cannot put a line break before the `(` in a function call. This restriction avoids some ambiguities in the language. If you write

```
a = f
(g).x(a)
```

Lua would see that as a single statement, `a = f(g).x(a)`. So, if you want two statements, you must add a semi-colon between them. If you actually want to call `f`, you must remove the line break before `(g)`.

lua-tailcall

A call of the form `return functioncall` is called a tail call. Lua implements proper tail calls (or proper tail recursion): in a tail call, the called function reuses the stack entry of the calling function. Therefore, there is no limit on the number of nested tail calls that a program can execute. However, a tail call erases any debug information about the calling function. Note that a tail call only happens with a particular syntax, where the `return` has one single function call as argument; this syntax makes the calling function return exactly the returns of the called function. So, none of the following examples are tail calls:

```
return (f(x))        -- results adjusted to 1
return 2 * f(x)
return x, f(x)       -- additional results
f(x); return         -- results discarded
return x or f(x)     -- results adjusted to 1
```

2.5.9 Function Definitions

lua-function-define

The syntax for function definition is

```
function ::= function funcbody
funcbody ::= ( [ parlist1 ] ) block end
```

The following syntactic sugar simplifies function definitions:

```
stat ::= function funcname funcbody
stat ::= local function Name funcbody
funcname ::= Name { . Name } [ : Name ]
```

The statement

`function f () body end`

translates to

`f = function () body end`

The statement

`function t.a.b.c.f () body end`

translates to

`t.a.b.c.f = function () body end`

The statement

`local function f () body end`

translates to

`local f; f = function f () body end`

not to

`local f = function f () body end`

(This only makes a difference when the body of the function contains references to `f`.)

lua-closure

A function definition is an executable expression, whose value has type `function`. When Lua pre-compiles a chunk, all its function bodies are pre-compiled too. Then, whenever Lua executes the function definition, the function is instantiated (or closed). This function instance (or closure) is the final value of the expression. Different instances of the same function may refer to different external local variables and may have different environment tables.

Parameters act as local variables that are initialized with the argument values:

```
parlist1 ::= namelist [ , ... ] | ...
```

lua-vararg

When a function is called, the list of arguments is adjusted to the length of the list of parameters, unless the function is a variadic or vararg function, which is indicated by three dots (`...`) at the end of its parameter list. A vararg function does not adjust its argument list; instead, it collects all extra arguments and supplies them to the function through a vararg expression, which is also written as three dots. The value of this expression is a list of all actual extra arguments, similar to a function with multiple results. If a vararg expression is used inside another expression or in the middle of a list of expressions, then its return list is adjusted to one element. If the expression is used as the last element of a list of expressions, then no adjustment is made (unless the call is enclosed in parentheses).

As an example, consider the following definitions:

```
function f(a, b) end
function g(a, b, ...) end
function r() return 1,2,3 end
```

Then, we have the following mapping from arguments to parameters and to the vararg expression:

```
CALL            PARAMETERS

f(3)             a=3, b=nil
f(3, 4)          a=3, b=4
f(3, 4, 5)       a=3, b=4
f(r(), 10)       a=1, b=10
f(r())           a=1, b=2

g(3)             a=3, b=nil, ... -->  (nothing)
g(3, 4)          a=3, b=4,   ... -->  (nothing)
g(3, 4, 5, 8)    a=3, b=4,   ... -->  5  8
g(5, r())        a=5, b=1,   ... -->  2  3
```

Results are returned using the `return` statement (see lua-control). If control reaches the end of a function without encountering a `return` statement, then the function returns with no results.

lua-colonsyntax

The colon syntax is used for defining methods, that is, functions that have an implicit extra parameter `self`. Thus, the statement

`function t.a.b.c:f ( params ) body end`

is syntactic sugar for

`t.a.b.c:f = function ( self, params ) body end`

2.6 Visibility Rules **lua-visibility**

Lua is a lexically scoped language. The scope of variables begins at the first statement after their declaration and lasts until the end of the innermost block that includes the declaration. Consider the following example:

```
x = 10                -- global variable
do                    -- new block
  local x = x         -- new `x`, with value 10
  print(x)            --> 10
  x = x+1
  do                  -- another block
    local x = x+1     -- another `x`
    print(x)          --> 12
  end
  print(x)            --> 11
end
print(x)              --> 10  (the global one)
```

Notice that, in a declaration like `local x = x`, the new x being declared is not in scope yet, and so the second x refers to the outside variable.

lua-upvalue

Because of the lexical scoping rules, local variables can be freely accessed by functions defined inside their scope. A local variable used by an inner function is called an upvalue, or external local variable, inside the inner function.

Notice that each execution of a local statement defines new local variables. Consider the following example:

```
a = {}
local x = 20
for i=1,10 do
  local y = 0
  a[i] = function () y=y+1; return x+y end
end
```

The loop creates ten closures (that is, ten instances of the anonymous function). Each of these closures uses a different y variable, while all of them share the same x.

2.7 Error Handling **lua-errors**

Because Lua is an embedded extension language, all Lua actions start from C code in the host program calling a function from the Lua library (see lua_pcall()). Whenever an error occurs during Lua compilation or execution, control returns to C, which can take appropriate measures (such as print an error message).

Lua code can explicitly generate an error by calling the `error` function (see error()). If you need to catch errors in Lua, you can use the `pcall` function (see pcall()).

2.8 Metatables **metatable** **lua-metatable**

Every value in Lua may have a metatable. This metatable is an ordinary Lua table that defines the behavior of the original table and userdata under certain special operations. You can change several aspects of the behavior of an object by setting specific fields in its metatable. For instance, when a non-numeric value is the operand of an addition, Lua checks for a function in the field "__add" in its metatable. If it finds one, Lua calls that function to perform the addition.

We call the keys in a metatable events and the values metamethods. In the previous example, the event is "add" and the metamethod is the function that performs the addition.

You can query the metatable of any value through the `getmetatable` function (see getmetatable()).

You can replace the metatable of tables through the `setmetatable` function (see setmetatable()). You cannot change the metatable of other types from Lua (except using the debug library); you must use the C API for that.

Tables and userdata have individual metatables (although multiple tables and userdata can share a same table as their metatable); values of all other types share one single metatable per type. So, there is one single metatable for all numbers, and for all strings, etc.

A metatable may control how an object behaves in arithmetic operations, order comparisons, concatenation, length operation, and indexing. A metatable can also define a function to be called when a userdata is garbage collected. For each of those operations Lua associates a specific key called an event. When Lua performs one of those operations over a value, it checks whether this value has a metatable with the corresponding event. If so, the value associated with that key (the metamethod) controls how Lua will perform the operation.

Metatables control the operations listed next. Each operation is identified by its corresponding name. The key for each operation is a string with its name prefixed by two underscores, __; for instance, the key for operation "add" is the string "__add". The semantics of these operations is better explained by a Lua function describing how the interpreter executes that operation.

The code shown here in Lua is only illustrative; the real behavior is hard coded in the interpreter and it is much more efficient than this simulation. All functions used in these descriptions (`rawget`, `tonumber`, etc.) are described in lua-lib-core. In particular, to retrieve the metamethod of a given object, we use the expression

```
metatable(obj)[event]
```

This should be read as

```
rawget(metatable(obj) or {}, event)
```

That is, the access to a metamethod does not invoke other metamethods, and the access to objects with no metatables does not fail (it simply results in `nil`).

"add": __add()
------ the + operation.

The function `getbinhandler` below defines how Lua chooses a handler for a binary operation. First, Lua tries the first operand. If its type does not define a handler for the operation, then Lua tries the second operand.

```
function getbinhandler (op1, op2, event)
  return metatable(op1)[event] or metatable(op2)[event]
end
```

By using this function, the behavior of the `op1 + op2` is

```
function add_event (op1, op2)
  local o1, o2 = tonumber(op1), tonumber(op2)
  if o1 and o2 then  -- both operands are numeric?
    return o1 + o2   -- `+` here is the primitive `add`
  else  -- at least one of the operands is not numeric
    local h = getbinhandler(op1, op2, "__add")
    if h then
      -- call the handler with both operands
      return h(op1, op2)
    else  -- no handler available: default behavior
      error(...)
    end
  end
end
```

"sub": __sub()
------ the - operation. Behavior similar to the "add" operation.

"mul": __mul()
------ the * operation. Behavior similar to the "add" operation.

"div": __div()
------ the / operation. Behavior similar to the "add" operation.

"mod": __mod()
------ the % operation. Behavior similar to the "add" operation, with the operation `o1 - floor(o1/o2)*o2` as the primitive operation.

"pow": __pow()
------ the ^ (exponentiation) operation. Behavior similar to the "add" operation, with the function pow (from the C math library) as the primitive operation.

"unm": __unm()
------ the unary - operation.

```
function unm_event (op)
  local o = tonumber(op)
  if o then  -- operand is numeric?
    return -o  -- `-` here is the primitive `unm`
  else  -- the operand is not numeric.
    -- Try to get a handler from the operand
    local h = metatable(op).__unm
    if h then
      -- call the handler with the operand
      return h(op)
    else  -- no handler available: default behavior
      error(...)
    end
  end
end
```

"concat": __concat()
--------- the `..` (concatenation) operation.

```
function concat_event (op1, op2)
  if (type(op1) == "string" or type(op1) == "number") and
     (type(op2) == "string" or type(op2) == "number") then
    return op1 .. op2  -- primitive string concatenation
  else
    local h = getbinhandler(op1, op2, "__concat")
    if h then
      return h(op1, op2)
    else
      error(...)
    end
  end
end
```

"len": __len()
------ the # operation.

```
function len_event (op)
  if type(op) == "string" then
    return strlen(op)         -- primitive string length
  elseif type(op) == "table" then
    return #op                -- primitive table length
  else
    local h = metatable(op).__len
    if h then
      -- call the handler with the operand
      return h(op)
    else  -- no handler available: default behavior
      error(...)
    end
  end
end
```

"eq": __eq()
----- the == operation.

The function `getcomphandler` defines how Lua chooses a metamethod for comparison operators. A metamethod only is selected when both objects being compared have the same type and the same metamethod for the selected operation.

```
function getcomphandler (op1, op2, event)
  if type(op1) ~= type(op2) then return nil end
  local mm1 = metatable(op1)[event]
  local mm2 = metatable(op2)[event]
  if mm1 == mm2 then return mm1 else return nil end
end
```

The "eq" event is defined as follows:

```
function eq_event (op1, op2)
  if type(op1) ~= type(op2) then  -- different types?
    return false   -- different objects
  end
  if op1 == op2 then   -- primitive equal?
    return true   -- objects are equal
  end
  -- try metamethod
  local h = getcomphandler(op1, op2, "__eq")
  if h then
    return h(op1, op2)
  else
    return false
  end
end
```

`a ~= b` is equivalent to `not (a == b)`.

"lt": __lt()
----- the < operation.

```
function lt_event (op1, op2)
  if type(op1) == "number" and type(op2) == "number" then
    return op1 < op2   -- numeric comparison
  elseif type(op1) == "string" and type(op2) == "string" then
    return op1 < op2   -- lexicographic comparison
  else
    local h = getcomphandler(op1, op2, "__lt")
    if h then
      return h(op1, op2)
    else
      error(...);
    end
  end
end
```

`a > b` is equivalent to `b < a`.

"le": __le()
----- the <= operation.

```
function le_event (op1, op2)
  if type(op1) == "number" and type(op2) == "number" then
    return op1 <= op2   -- numeric comparison
  elseif type(op1) == "string" and type(op2) == "string" then
    return op1 <= op2   -- lexicographic comparison
  else
    local h = getcomphandler(op1, op2, "__le")
    if h then
      return h(op1, op2)
    else
      h = getcomphandler(op1, op2, "__lt")
      if h then
        return not h(op2, op1)
      else
        error(...);
      end
    end
  end
end
```

`a >= b` is equivalent to `b <= a`. Note that, in the absence of a "le" metamethod, Lua tries the "lt", assuming that `a <= b` is equivalent to `not (b < a)`.

"index": __index()
-------- The indexing access `table[key]`.

```
function gettable_event (table, key)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    if v ~= nil then return v end
    h = metatable(table).__index
    if h == nil then return nil end
  else
    h = metatable(table).__index
    if h == nil then
      error(...);
    end
  end
  if type(h) == "function" then
    return h(table, key)      -- call the handler
  else return h[key]          -- or repeat operation on it
end
```

"newindex": __newindex()
---------- The indexing assignment `table[key] = value`.

```
function settable_event (table, key, value)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    if v ~= nil then rawset(table, key, value); return end
    h = metatable(table).__newindex
    if h == nil then rawset(table, key, value); return end
  else
    h = metatable(table).__newindex
    if h == nil then
      error(...);
    end
  end
  if type(h) == "function" then
    return h(table, key,value)    -- call the handler
  else h[key] = value             -- or repeat operation on it
end
```

"call": __call()
------- called when Lua calls a value.

```
function function_event (func, ...)
  if type(func) == "function" then
    return func(...)   -- primitive call
  else
    local h = metatable(func).__call
    if h then
      return h(func, ...)
    else
      error(...)
    end
  end
end
```

2.9 Environments

lua-environments

Besides metatables, objects of types thread, function, and userdata have another table associated with them, called their environment. Like metatables, environments are regular tables and multiple objects can share the same environment.

Environments associated with userdata have no meaning for Lua. It is only a convenience feature for programmers to associate a table to a userdata.

Environments associated with threads are called global environments. They are used as the default environment for their threads and non-nested functions created by the thread (through loadfile(), loadstring() or load()) and can be directly accessed by C code (see lua-pseudoindex).

Environments associated with C functions can be directly accessed by C code (see lua-pseudoindex). They are used as the default environment for other C functions created by the function.

Environments associated with Lua functions are used to resolve all accesses to global variables within the function (see lua-variables). They are used as the default environment for other Lua functions created by the function.

You can change the environment of a Lua function or the running thread by calling `setfenv`. You can get the environment of a Lua function or the running thread by calling `getfenv` (see lua_getfenv()). To manipulate the environment of other objects (userdata, C functions, other threads) you must use the C API.

2.10 Garbage Collection

lua-gc

Lua performs automatic memory management. This means that you do not have to worry neither about allocating memory for new objects nor about freeing it when the objects are no longer needed. Lua manages memory automatically by running a garbage collector from time to time to collect all dead objects (that is, these objects that are no longer accessible from Lua). All objects in Lua are subject to automatic management: tables, userdata, functions, threads, and strings.

Lua implements an incremental mark-and-sweep collector. It uses two numbers to control its garbage-collection cycles: the garbage-collector pause and the garbage-collector step multiplier.

The garbage-collector pause controls how long the collector waits before starting a new cycle. Larger values make the collector less aggressive. Values smaller than 1 mean the collector will not wait to start a new cycle. A value of 2 means that the collector waits for the total memory in use to double before starting a new cycle.

The step multiplier controls the relative speed of the collector relative to memory allocation. Larger values make the collector more aggressive but also increase the size of each incremental step. Values smaller than 1 make the collector too slow and may result in the collector never finishing a cycle. The default, 2, means that the collector runs at "twice" the speed of memory allocation.

You can change these numbers by calling `lua_gc` (see lua_gc()) in C or `collectgarbage` (see collectgarbage()) in Lua. Both get percentage points as arguments (so an argument of 100 means a real value of 1). With these functions you can also control the collector directly (e.g., stop and restart it).

2.10.1 Garbage-Collection Metamethods

lua-gc-meta

Using the C API, you can set garbage-collector metamethods for userdata (see lua-metatable). These metamethods are also called finalizers. Finalizers allow you to coordinate Lua's garbage collection with external resource management (such as closing files, network or database connections, or freeing your own memory).

__gc

Garbage userdata with a field `__gc` in their metatables are not collected immediately by the garbage collector. Instead, Lua puts them in a list. After the collection, Lua does the equivalent of the following function for each userdata in that list:

```
function gc_event (udata)
  local h = metatable(udata).__gc
  if h then
    h(udata)
  end
end
```

At the end of each garbage-collection cycle, the finalizers for userdata are called in reverse order of their creation, among these collected in that cycle. That is, the first finalizer to be called is the one associated with the userdata created last in the program.

2.10.2 - Weak Tables

lua-weaktable

A weak table is a table whose elements are weak references. A weak reference is ignored by the garbage collector. In other words, if the only references to an object are weak references, then the garbage collector will collect this object.

__mode

A weak table can have weak keys, weak values, or both. A table with weak keys allows the collection of its keys, but prevents the collection of its values. A table with both weak keys and weak values allows the collection of both keys and values. In any case, if either the key or the value is collected, the whole pair is removed from the table. The weakness of a table is controlled by the value of the `__mode` field of its metatable. If the `__mode` field is a string containing the character `k`, the keys in the table are weak. If `__mode` contains `v`, the values in the table are weak.

After you use a table as a metatable, you should not change the value of its field `__mode`. Otherwise, the weak behavior of the tables controlled by this metatable is undefined.

2.11 Coroutines

lua-coroutine

Lua supports coroutines, also called collaborative multithreading. A coroutine in Lua represents an independent thread of execution. Unlike threads in multithread systems, however, a coroutine only suspends its execution by explicitly calling a yield function.

You create a coroutine with a call to `coroutine.create` (see coroutine.create()). Its sole argument is a function that is the main function of the coroutine. The `create` function only creates a new coroutine and returns a handle to it (an object of type `thread`); it does not start the coroutine execution.

When you first call `coroutine.resume` (see coroutine.resume()), passing as its first argument the thread returned by `coroutine.create`, the coroutine starts its execution, at the first line of its main function. Extra arguments passed to `coroutine.resume` are passed on to the

coroutine main function. After the coroutine starts running, it runs until it terminates or `yields`.

A coroutine can terminate its execution in two ways: normally, when its main function returns (explicitly or implicitly, after the last instruction); and abnormally, if there is an unprotected error. In the first case, `coroutine.resume` returns `true`, plus any values returned by the coroutine main function. In case of errors, `coroutine.resume` returns `false` plus an error message.

A coroutine yields by calling `coroutine.yield` (see coroutine.yield()). When a coroutine yields, the corresponding `coroutine.resume` returns immediately, even if the yield happens inside nested function calls (that is, not in the main function, but in a function directly or indirectly called by the main function). In the case of a yield, `coroutine.resume` also returns `true`, plus any values passed to `coroutine.yield`. The next time you resume the same coroutine, it continues its execution from the point where it yielded, with the call to `coroutine.yield` returning any extra arguments passed to `coroutine.resume`.

Like `coroutine.create`, the `coroutine.wrap` function (see coroutine.wrap()) also creates a coroutine, but instead of returning the coroutine itself, it returns a function that, when called, resumes the coroutine. Any arguments passed to this function go as extra arguments to `coroutine.resume`. `coroutine.wrap` returns all the values returned by `coroutine.resume`, except the first one (the boolean error code). Unlike `coroutine.resume`, `coroutine.wrap` does not catch errors; any error is propagated to the caller.

As an example, consider the next code:

```
function foo1 (a)
  print("foo", a)
  return coroutine.yield(2*a)
end

co = coroutine.create(function (a,b)
      print("co-body", a, b)
      local r = foo1(a+1)
      print("co-body", r)
      local r, s = coroutine.yield(a+b, a-b)
      print("co-body", r, s)
      return b, "end"
end)

print("main", coroutine.resume(co, 1, 10))
print("main", coroutine.resume(co, "r"))
print("main", coroutine.resume(co, "x", "y"))
print("main", coroutine.resume(co, "x", "y"))
```

When you run it, it produces the following output:

```
co-body 1       10
foo     2
main    true    4
co-body r
main    true    11      -9
co-body x       y
main    true    10      end
main    false   cannot resume dead coroutine
```

3 THE APPLICATION PROGRAM INTERFACE

lua-API

This section describes the C API for Lua, that is, the set of C functions available to the host program to communicate with Lua. All API functions and related types and constants are declared in the header file `lua.h`.

Even when we use the term "function", any facility in the API may be provided as a `macro` instead. All such macros use each of its arguments exactly once (except for the first argument, which is always a Lua state), and so do not generate hidden side-effects.

As in most C libraries, the Lua API functions do not check their arguments for validity or consistency. However, you can change this behavior by compiling Lua with a proper definition for the macro `luai_apicheck`,in file `luaconf.h`.

3.1 The Stack

lua-stack lua-apiStack

Lua uses a virtual stack to pass values to and from C. Each element in this stack represents a Lua value (`nil`, number, string, etc.).

Whenever Lua calls C, the called function gets a new stack, which is independent of previous stacks and of stacks of C functions that are still active. This stack initially contains any arguments to the C function and it is where the C function pushes its results to be returned to the caller (see lua_CFunction).

lua-stackindex

For convenience, most query operations in the API do not follow a strict stack discipline. Instead, they can refer to any element in the stack by using an index: a positive index represents an absolute stack position (starting at 1); a negative index represents an offset from the top of the stack. More specifically, if the stack has n elements, then index 1 represents the first element (that is, the element that was pushed onto the stack first) and index n represents the last element; index `-1` also represents the last element (that is, the element at the top) and index `-n` represents the first element. We say that an index is valid if it lies between 1 and the stack top (that is, if `1 <= abs(index) <= top`).

3.2 Stack Size

lua-apiStackSize

When you interact with Lua API, you are responsible for ensuring consistency. In particular, you are responsible for controlling stack overflow. You can use the function `lua_checkstack` to grow the stack size (see lua_checkstack()).

Whenever Lua calls C, it ensures that at least `LUA_MINSTACK` stack positions are available. `LUA_MINSTACK` is defined as 20, so that usually you do not have to worry about stack space unless your code has loops pushing elements onto the stack.

Most query functions accept as indices any value inside the available stack space, that is, indices up to the maximum stack size you have set through `lua_checkstack`. Such indices are called acceptable indices. More formally, we define an acceptable index as follows:

```
(index < 0 && abs(index) <= top) || (index > 0 && index <= stackspace)
```

Note that 0 is never an acceptable index.

3.3 Pseudo-Indices lua-pseudoindex

Unless otherwise noted, any function that accepts valid indices can also be called with pseudo-indices, which represent some Lua values that are accessible to the C code but which are not in the stack. Pseudo-indices are used to access the thread environment, the function environment, the registry, and the upvalues of a C function (see lua-cclosure).

The thread environment (where global variables live) is always at pseudo-index `LUA_GLOBALSINDEX`. The environment of the running C function is always at pseudo-index `LUA_ENVIRONINDEX`.

To access and change the value of global variables, you can use regular table operations over an environment table. For instance, to access the value of a global variable, do

```
lua_getfield(L, LUA_GLOBALSINDEX, varname);
```

3.4 C Closures lua-cclosure

When a C function is created, it is possible to associate some values with it, thus creating a C closure; these values are called upvalues and are accessible to the function whenever it is called (see lua_pushcclosure()).

Whenever a C function is called, its upvalues are located at specific pseudo-indices. These pseudo-indices are produced by the macro `lua_upvalueindex`. The first value associated with a function is at position `lua_upvalueindex(1)`, and so on. Any access to `lua_upvalueindex( n )`, where n is greater than the number of upvalues of the current function, produces an acceptable (but invalid) index.

3.5 Registry lua-registry

Lua provides a registry, a pre-defined table that can be used by any C code to store whatever Lua value it needs to store. This table is always located at pseudo-index `LUA_REGISTRYINDEX`. Any C library can store data into this table, but it should take care to choose keys different from those used by other libraries, to avoid collisions. Typically, you should use as key a string containing your library name or a light userdata with the address of a C object in your code.

The integer keys in the registry are used by the reference mechanism, implemented by the auxiliary library, and therefore should not be used for other purposes.

3.6 Error Handling in C lua-apiError

Internally, Lua uses the C `longjmp` facility to handle errors. (You can also choose to use exceptions if you use C++; see file `luaconf.h`.) When Lua faces any error (such as memory allocation errors, type errors, syntax errors, and runtime errors) it raises an error; that is, it does a long jump. A protected environment uses `setjmp` to set a recover point; any error jumps to the most recent active recover point.

Almost any function in the API may raise an error, for instance due to a memory allocation error. The following functions run in protected mode (that is, they create a protected environment to run), so they never raise an error: `lua_newstate`, `lua_close`, `lua_load`, `lua_pcall`, and `lua_cpcall` (see lua_newstate(), lua_close(), lua_load(), lua_pcall(), and lua_cpcall()).

Inside a C function you can raise an error by calling `lua_error` (see lua_error()).

3.7 Functions and Types lua-apiFunctions

Here we list all functions and types from the C API in alphabetical order.

lua_Alloc lua_Alloc

```
typedef void * (*lua_Alloc) (void *ud,
                             void *ptr,
                             size_t osize,
                             size_t nsize);
```

The type of the memory-allocation function used by Lua states. The allocator function must provide a functionality similar to `realloc`, but not exactly the same. Its arguments are ud, an opaque pointer passed to `lua_newstate` (see lua_newstate()); `ptr`, a pointer to the block being allocated/reallocated/freed; `osize`, the original size of the block; `nsize`, the new size of the block. `ptr` is `NULL` if and only if `osize` is zero. When `nsize` is zero, the allocator must return `NULL`; if `osize` is not zero, it should free the block pointed to by `ptr`. When `nsize` is not zero, the allocator returns `NULL` if and only if it cannot fill the request. When `nsize` is not zero and `osize` is zero, the allocator should behave like `malloc`. When `nsize` and `osize` are not zero, the allocator behaves like `realloc`. Lua assumes that the allocator never fails when `osize >= nsize`.

Here is a simple implementation for the allocator function. It is used in the auxiliary library by `luaL_newstate` (see luaL_newstate()).

```
static void *l_alloc (void *ud, void *ptr, size_t osize,
                                           size_t nsize) {
  (void)ud;  (void)osize;  /* not used */
  if (nsize == 0) {
    free(ptr);
    return NULL;
  }
  else
    return realloc(ptr, nsize);
}
```

This code assumes that `free(NULL)` has no effect and that `realloc(NULL, size)` is equivalent to `malloc(size)`. ANSI C ensures both behaviors.

lua_atpanic lua_atpanic()

```
lua_CFunction lua_atpanic (lua_State *L, lua_CFunction panicf);
```

Sets a new panic function and returns the old one.

If an error happens outside any protected environment, Lua calls a `panic function` and then calls `exit(EXIT_FAILURE)`, thus exiting the host application. Your panic function may avoid this exit by never returning (e.g., doing a long jump).

The panic function can access the error message at the top of the stack.

lua_call lua_call()

```
void lua_call (lua_State *L, int nargs, int nresults);
```

Calls a function.

To call a function you must use the following protocol: first, the function to be called is pushed onto the stack; then, the arguments to the function are pushed in direct order; that is, the first argument is pushed first. Finally you call `lua_call`; `nargs` is the number of arguments that you pushed onto the stack. All arguments and the function value are popped from the stack when the function is called. The function results are pushed onto the stack when the function returns. The number of results is adjusted to `nresults`, unless `nresults` is `LUA_MULTRET`. In this case, `all` results from the function are pushed. Lua takes care that the returned values fit into the stack space. The function results are pushed onto the stack in direct order (the first result is pushed first), so that after the call the last result is on the top of the stack.

Any error inside the called function is propagated upwards (with a `longjmp`).

The following example shows how the host program may do the equivalent to this Lua code:

```
a = f("how", t.x, 14)
```

Here it is in C:

```
lua_getfield(L, LUA_GLOBALSINDEX, "f"); // function to be called
lua_pushstring(L, "how");                       // 1st argument
lua_getfield(L, LUA_GLOBALSINDEX, "t");   // table to be indexed
lua_getfield(L, -1, "x");        // push result of t.x (2nd arg)
lua_remove(L, -2);                  // remove 't' from the stack
lua_pushinteger(L, 14);                         // 3rd argument
lua_call(L, 3, 1);     // call 'f' with 3 arguments and 1 result
lua_setfield(L, LUA_GLOBALSINDEX, "a");        // set global 'a'
```

Note that the code above is "balanced": at its end, the stack is back to its original configuration. This is considered good programming practice.

lua_CFunction lua-cfunction lua_CFunction

```
typedef int (*lua_CFunction) (lua_State *L);
```

Type for C functions.

In order to communicate properly with Lua, a C function must use the following protocol, which defines the way parameters and results are passed: a C function receives its arguments from Lua in its stack in direct order (the first argument is pushed first). So, when the function starts, `lua_gettop(L)` (see lua_gettop()) returns the number of arguments received by the function. The first argument (if any) is at index 1 and its last argument is at index `lua_gettop(L)`. To return values to Lua, a C function just pushes them onto the stack, in direct order (the first result is pushed first), and returns the number of results. Any other value in the stack below the results will be properly discarded by Lua. Like a Lua function, a C function called by Lua can also return many results.

lua-cfunctionexample

As an example, the following function receives a variable number of numerical arguments and returns their average and sum:

```
static int foo (lua_State *L) {
  int n = lua_gettop(L);    /* number of arguments */
  lua_Number sum = 0;
  int i;
  for (i = 1; i &lt;= n; i++) {
    if (!lua_isnumber(L, i)) {
      lua_pushstring(L, "incorrect argument");
      lua_error(L);
    }
    sum += lua_tonumber(L, i);
  }
  lua_pushnumber(L, sum/n); /* first result */
  lua_pushnumber(L, sum);   /* second result */
  return 2;                 /* number of results */
}
```

lua_checkstack lua_checkstack()

```
int lua_checkstack (lua_State *L, int extra);
```

Ensures that there are at least `extra` free stack slots in the stack. It returns false if it cannot grow the stack to that size. This function never shrinks the stack; if the stack is already larger than the new size, it is left unchanged.

lua_close lua_close()

```
void lua_close (lua_State *L);
```

Destroys all objects in the given Lua state (calling the corresponding garbage-collection metamethods, if any) and frees all dynamic memory used by this state. On several platforms, you may not need to call this function, because all resources are naturally released when the host

program ends. On the other hand, long-running programs, such as a daemon or a web server, might need to release states as soon as they are not needed, to avoid growing too large.

lua_concat lua_concat()

```
void lua_concat (lua_State *L, int n);
```

Concatenates the n values at the top of the stack, pops them, and leaves the result at the top. If n is 1, the result is the single string on the stack (that is, the function does nothing); if n is 0, the result is the empty string. Concatenation is done following the usual semantics of Lua (see lua-concat).

lua_cpcall lua_cpcall()

```
int lua_cpcall (lua_State *L, lua_CFunction func, void *ud);
```

Calls the C function `func` in protected mode. `func` starts with only one element in its stack, a light userdata containing `ud`. In case of errors, `lua_cpcall` returns the same error codes as `lua_pcall` (see lua_pcall()), plus the error object on the top of the stack; otherwise, it returns zero, and does not change the stack. All values returned by `func` are discarded.

lua_createtable lua_createtable()

```
void lua_createtable (lua_State *L, int narr, int nrec);
```

Creates a new empty table and pushes it onto the stack. The new table has space pre-allocated for `narr` array elements and `nrec` non-array elements. This pre-allocation is useful when you know exactly how many elements the table will have. Otherwise you can use the function `lua_newtable` (see lua_newtable()).

lua_dump lua_dump()

```
int lua_dump (lua_State *L, lua_Writer writer, void *data);
```

Dumps a function as a binary chunk. Receives a Lua function on the top of the stack and produces a binary chunk that, if loaded again, results in a function equivalent to the one dumped. As it produces parts of the chunk, `lua_dump` calls function `writer` (see lua_Writer) with the given `data` to write them.

The value returned is the error code returned by the last call to the writer; 0 means no errors.

This function does not pop the Lua function from the stack.

lua_equal lua_equal()

```
int lua_equal (lua_State *L, int index1, int index2);
```

Returns 1 if the two values in acceptable indices `index1` and `index2` are equal, following the semantics of the Lua == operator (that is, may call metamethods). Otherwise returns 0. Also returns 0 if any of the indices is non valid.

lua_error lua_error()

```
int lua_error (lua_State *L);
```

Generates a Lua error. The error message (which can actually be a Lua value of any type) must be on the stack top. This function does a long jump, and therefore never returns (see luaL_error()).

lua_gc lua_gc()

```
int lua_gc (lua_State *L, int what, int data);
```

Controls the garbage collector.

This function performs several tasks, according to the value of the parameter `what`:

- `LUA_GCSTOP` stops the garbage collector.
- `LUA_GCRESTART` restarts the garbage collector.
- `LUA_GCCOLLECT` performs a full garbage-collection cycle.
- `LUA_GCCOUNT` returns the current amount of memory (in Kbytes) in use by Lua.
- `LUA_GCCOUNTB` returns the remainder of dividing the current amount of bytes of memory in use by Lua by 1024.
- `LUA_GCSTEP` performs an incremental step of garbage collection. The step "size" is controlled by `data` (larger values mean more steps) in a non-specified way. If you want to control the step size you must experimentally tune the value of `data`. The function returns 1 if the step finished a garbage-collection cycle.
- `LUA_GCSETPAUSE` sets `data` /100 as the new value for the `pause` of the collector (see lua-gc). The function returns the previous value of the pause.
- `LUA_GCSETSTEPMUL`sets `data` /100 as the new value for the `step multiplier` of the collector (see lua-gc). The function returns the previous value of the step multiplier.

lua_getallocf lua_getallocf()

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
```

Returns the memory-allocation function of a given state. If ud is not `NULL`, Lua stores in *ud the opaque pointer passed to `lua_newstate` (see lua_newstate()).

lua_getfenv lua_getfenv()

```
void lua_getfenv (lua_State *L, int index);
```

Pushes onto the stack the environment table of the value at the given index.

lua_getfield lua_getfield()

```
void lua_getfield (lua_State *L, int index, const char *k);
```

Pushes onto the stack the value `t[k]`, where `t` is the value at the given valid index `index`. As in Lua, this function may trigger a metamethod for the "index" event (see lua-metatable).

lua_getglobal lua_getglobal()

```
void lua_getglobal (lua_State *L, const char *name);
```

Pushes onto the stack the value of the global `name`. It is defined as a macro:

```
#define lua_getglobal(L,s)  lua_getfield(L, LUA_GLOBALSINDEX, s)
```

lua_getmetatable lua_getmetatable()

```
int lua_getmetatable (lua_State *L, int index);
```

Pushes onto the stack the metatable of the value at the given acceptable index. If the index is not valid, or if the value does not have a metatable, the function returns 0 and pushes nothing on the stack.

lua_gettable lua_gettable()

```
void lua_gettable (lua_State *L, int index);
```

Pushes onto the stack the value `t[k]`, where `t` is the value at the given valid index `index` and `k` is the value at the top of the stack.

This function pops the key from the stack (putting the resulting value in its place). As in Lua, this function may trigger a metamethod for the "index" event (see lua-metatable).

lua_gettop lua_gettop()

```
int lua_gettop (lua_State *L);
```

Returns the index of the top element in the stack. Because indices start at 1, this result is equal to the number of elements in the stack (and so 0 means an empty stack).

lua_insert lua_insert()

```
void lua_insert (lua_State *L, int index);
```

Moves the top element into the given valid index, shifting up the elements above this index to open space. Cannot be called with a pseudo-index, because a pseudo-index is not an actual stack position.

lua_Integer lua_Integer

```
typedef ptrdiff_t lua_Integer;
```

The type used by the Lua API to represent integral values.

By default it is a `ptrdiff_t`, which is usually the largest integral type the machine handles "comfortably".

lua_isboolean lua_isboolean()

```
int lua_isboolean (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index has type boolean, and 0 otherwise.

lua_iscfunction lua_iscfunction()

```
int lua_iscfunction (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a C function, and 0 otherwise.

lua_isfunction lua_isfunction()

```
int lua_isfunction (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a function (either C or Lua), and 0 otherwise.

lua_islightuserdata lua_islightuserdata()

```
int lua_islightuserdata (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a light userdata, and 0 otherwise.

lua_isnil lua_isnil()

```
int lua_isnil (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is `nil`, and 0 otherwise.

lua_isnumber lua_isnumber()

```
int lua_isnumber (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a number or a string convertible to a number, and 0 otherwise.

lua_isstring lua_isstring()

```
int lua_isstring (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a string or a number (which is always convertible to a string), and 0 otherwise.

lua_istable lua_istable()

```
int lua_istable (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a table, and 0 otherwise.

lua_isthread lua_isthread()

```
int lua_isthread (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a thread, and 0 otherwise.

lua_isuserdata lua_isuserdata()

```
int lua_isuserdata (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a userdata (either full or light), and 0 otherwise.

lua_lessthan lua_lessthan()

```
int lua_lessthan (lua_State *L, int index1, int index2);
```

Returns 1 if the value at acceptable index `index1` is smaller than the value at acceptable index `index2`, following the semantics of the Lua < operator (that is, may call metamethods). Otherwise returns 0. Also returns 0 if any of the indices is non valid.

lua_load lua_load()

```
int lua_load (lua_State *L,
              lua_Reader reader,
              void *data,
              const char *chunkname);
```

Loads a Lua chunk. If there are no errors, `lua_load` pushes the compiled chunk as a Lua function on top of the stack. Otherwise, it pushes an error message. The return values of `lua_load` are:

- `0`: no errors;
- `LUA_ERRSYNTAX` : syntax error during pre-compilation;
- `LUA_ERRMEM` : memory allocation error.

This function only loads a chunk; it does not run it.

`lua_load` automatically detects whether the chunk is text or binary, and loads it accordingly (see program `luac`).

The `lua_load` function uses a user-supplied `reader` function to read the chunk (see lua_Reader). The `data` argument is an opaque value passed to the reader function.

The `chunkname` argument gives a name to the chunk, which is used for error messages and in debug information (see lua-apiDebug).

lua_newstate lua_newstate()

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```

Creates a new, independent state. Returns `NULL` if cannot create the state (due to lack of memory). The argument `f` is the allocator function; Lua does all memory allocation for this state through this function. The second argument, `ud`, is an opaque pointer that Lua simply passes to the allocator in every call.

lua_newtable lua_newtable()

```
void lua_newtable (lua_State *L);
```

Creates a new empty table and pushes it onto the stack. It is equivalent to `lua_createtable(L, 0, 0)` (see lua_createtable()).

lua_newthread lua_newthread()

```
lua_State *lua_newthread (lua_State *L);
```

Creates a new thread, pushes it on the stack, and returns a pointer to a `lua_State` (see lua_State) that represents this new thread. The new state returned by this function shares with the original state all global objects (such as tables), but has an independent execution stack.

There is no explicit function to close or to destroy a thread. Threads are subject to garbage collection, like any Lua object.

lua_newuserdata lua_newuserdata()

```
void *lua_newuserdata (lua_State *L, size_t size);
```

This function allocates a new block of memory with the given size, pushes onto the stack a new full userdata with the block address, and returns this address. userdata

Userdata represents C values in Lua. A full userdata represents a block of memory. It is an object (like a table): you must create it, it can have its own metatable, and you can detect when it is being collected. A full userdata is only equal to itself (under raw equality).

When Lua collects a full userdata with a `gc` metamethod, Lua calls the metamethod and marks the userdata as finalized. When this userdata is collected again then Lua frees its corresponding memory.

lua_next lua_next()

```
int lua_next (lua_State *L, int index);
```

Pops a key from the stack, and pushes a key-value pair from the table at the given index (the "next" pair after the given key). If there are no more elements in the table, then `lua_next` returns 0 (and pushes nothing).

lua-tabletraversal

A typical traversal looks like this:

```
/* table is in the stack at index 't' */
lua_pushnil(L);  /* first key */
while (lua_next(L, t) != 0) {
  /* uses 'key' (at index -2) and 'value' (at index -1) */
  printf("%s - %s\n",
         lua_typename(L, lua_type(L, -2)),
         lua_typename(L, lua_type(L, -1)));
  /* removes 'value'; keeps 'key' for next iteration */
  lua_pop(L, 1);
}
```

While traversing a table, do not call `lua_tolstring` (see lua_tolstring()) directly on a key, unless you know that the key is actually a string. Recall that `lua_tolstring` changes the value at the given index; this confuses the next call to `lua_next`.

lua_Number lua_Number

```
typedef double lua_Number;
```

The type of numbers in Lua. By default, it is double, but that can be changed in `luaconf.h`.

Through the configuration file you can change Lua to operate with another type for numbers (e.g., float or long).

lua_objlen lua_objlen()

```
size_t lua_objlen (lua_State *L, int index);
```

Returns the "length" of the value at the given acceptable index: for strings, this is the string length; for tables, this is the result of the length operator (#); for userdata, this is the size of the block of memory allocated for the userdata; for other values, it is 0.

lua_pcall lua_pcall()

```
lua_pcall (lua_State *L, int nargs, int nresults, int errfunc);
```

Calls a function in protected mode.

Both `nargs` and `nresults` have the same meaning as in `lua_call` (see lua_call()). If there are no errors during the call, `lua_pcall` behaves exactly like `lua_call`. However, if there is any error, `lua_pcall` catches it, pushes a single value on the stack (the error message), and returns an error code. Like `lua_call`, `lua_pcall` always removes the function and its arguments from the stack.

If `errfunc` is 0, then the error message returned on the stack is exactly the original error message. Otherwise, `errfunc` is the stack index of an `error handler function`. (In the current implementation, this index cannot be a pseudo-index.) In case of runtime errors, this function will be called with the error message and its return value will be the message returned on the stack by `lua_pcall`.

Typically, the error handler function is used to add more debug information to the error message, such as a stack traceback. Such information cannot be gathered after the return of `lua_pcall`, since by then the stack has unwound.

The `lua_pcall` function returns 0 in case of success or one of the following error codes (defined in `lua.h`):

- `LUA_ERRRUN` a runtime error.
- `LUA_ERRMEM` memory allocation error. For such errors, Lua does not call the error handler function.
- `LUA_ERRERR` error while running the error handler function.

lua_pop lua_pop()

```
void lua_pop (lua_State *L, int n);
```

Pops n elements from the stack.

lua_pushboolean lua_pushboolean()

```
void lua_pushboolean (lua_State *L, int b);
```

Pushes a boolean value with value b onto the stack.

lua_pushcclosure lua_pushcclosure()

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);
```

Pushes a new C closure onto the stack.

When a C function is created, it is possible to associate some values with it, thus creating a C closure (see lua-cclosure); these values are then accessible to the function whenever it is called. To associate values with a C function, first these values should be pushed onto the stack (when there are multiple values, the first value is pushed first). Then `lua_pushcclosure` is called to create and push the C function onto the stack, with the argument n telling how many values should be associated with the function. `lua_pushcclosure` also pops these values from the stack.

lua_pushcfunction lua_pushcfunction()

```
void lua_pushcfunction (lua_State *L, lua_CFunction f);
```

Pushes a C function onto the stack. This function receives a pointer to a C function and pushes onto the stack a Lua value of type `function` that, when called, invokes the corresponding C function.

Any function to be registered in Lua must follow the correct protocol to receive its parameters and return its results (see lua_CFunction).

`lua_pushcfunction` is defined as a macro:

```
#define lua_pushcfunction(L,f)  lua_pushcclosure(L,f,0)
```

lua_pushfstring lua_pushfstring()

```
const char *lua_pushfstring (lua_State *L, const char *fmt, ...);
```

Pushes onto the stack a formatted string and returns a pointer to this string. It is similar to the C function `sprintf`, but has some important differences:

- You do not have to allocate space for the result: the result is a Lua string and Lua takes care of memory allocation (and deallocation, through garbage collection).
- The conversion specifiers are quite restricted. There are no flags, widths, or precisions. The conversion specifiers can only be `%%` (inserts a `%` in the string), `%s` (inserts a zero-terminated string, with no size restrictions), `%f` (inserts a `lua_Number`), `%p` (inserts a pointer as a hexadecimal numeral), `%d` (inserts an `int`), and `%c` (inserts an `int` as a character).

lua_pushinteger lua_pushinteger()

```
void lua_pushinteger (lua_State *L, lua_Integer n);
```

Pushes a number with value n onto the stack.

lua_pushlightuserdata lua_pushlightuserdata()

```
void lua_pushlightuserdata (lua_State *L, void *p);
```

Pushes a light userdata onto the stack. lua-lightuserdata
Userdata represents C values in Lua. A light userdata represents a pointer. It is a value (like a number): you do not create it, it has no individual metatable, and it is not collected (as it was never created). A light userdata is equal to "any" light userdata with the same C address.

lua_pushlstring lua_pushlstring()

```
void lua_pushlstring (lua_State *L, const char *s, size_t len);
```

Pushes the string pointed to by `s` with size `len` onto the stack. Lua makes (or reuses) an internal copy of the given string, so the memory at `s` can be freed or reused immediately after the function returns. The string can contain embedded zeros.

lua_pushnil lua_pushnil()

```
void lua_pushnil (lua_State *L);
```

Pushes a nil value onto the stack.

lua_pushnumber lua_pushnumber()

```
void lua_pushnumber (lua_State *L, lua_Number n);
```

Pushes a number with value n onto the stack.

lua_pushstring lua_pushstring()

```
void lua_pushstring (lua_State *L, const char *s);
```

Pushes the zero-terminated string pointed to by `s` onto the stack. Lua makes (or reuses) an internal copy of the given string, so the memory at `s` can be freed or reused immediately after the function returns. The string cannot contain embedded zeros; it is assumed to end at the first zero.

lua_pushthread lua_pushthread()

```
int lua_pushthread (lua_State *L);
```

Pushes the thread represented by L onto the stack. Returns 1 if this thread is the main thread of its state.

lua_pushvalue lua_pushvalue()

```
void lua_pushvalue (lua_State *L, int index);
```

Pushes a copy of the element at the given valid index onto the stack.

lua_pushvfstring lua_pushvfstring()

```
const char *lua_pushvfstring (lua_State *L,
                              const char *fmt,
                              va_list argp);
```

Equivalent to `lua_pushfstring` (see lua_pushfstring()), except that it receives a `va_list` instead of a variable number of arguments.

lua_rawequal lua_rawequal()

```
int lua_rawequal (lua_State *L, int index1, int index2);
```

Returns 1 if the two values in acceptable indices `index1` and `index2` are primitively equal (that is, without calling metamethods). Otherwise returns 0. Also returns 0 if any of the indices are non valid.

lua_rawget lua_rawget()

```
void lua_rawget (lua_State *L, int index);
```

Similar to `lua_gettable` (see lua_gettable()), but does a raw access (i.e., without metamethods).

lua_rawgeti lua_rawgeti()

```
void lua_rawgeti (lua_State *L, int index, int n);
```

Pushes onto the stack the value `t[n]`, where `t` is the value at the given valid index `index`. The access is raw; that is, it does not invoke metamethods.

lua_rawset lua_rawset()

```
void lua_rawset (lua_State *L, int index);
```

Similar to `lua_settable` (see lua_settable()), but does a raw assignment (i.e., without metamethods).

lua_rawseti lua_rawseti()

```
void lua_rawseti (lua_State *L, int index, int n);
```

Does the equivalent of `t[n] = v`, where `t` is the value at the given valid index `index` and `v` is the value at the top of the stack.

This function pops the value from the stack. The assignment is raw; that is, it does not invoke metamethods.

lua_Reader lua_Reader

```
typedef const char * (*lua_Reader) (lua_State *L,
                                    void *data,
                                    size_t *size);
```

The reader function used by `lua_load` (see lua_load()). Every time it needs another piece of the chunk, `lua_load` calls the reader, passing along its `data` parameter. The reader must return a pointer to a block of memory with a new piece of the chunk and set `size` to the block size. The block must exist until the reader function is called again. To signal the end of the chunk, the reader must return `NULL`. The reader function may return pieces of any size greater than zero.

lua_register lua_register()

```
void lua_register (lua_State *L,
                   const char *name,
                   lua_CFunction f);
```

Sets the C function `f` as the new value of global `name`. It is defined as a macro:

```
#define lua_register(L,n,f) \
       (lua_pushcfunction(L, f), lua_setglobal(L, n))
```

lua_remove lua_remove()

```
void lua_remove (lua_State *L, int index);
```

Removes the element at the given valid index, shifting down the elements above this index to fill the gap. Cannot be called with a pseudo-index, because a pseudo-index is not an actual stack position.

lua_replace lua_replace()

```
void lua_replace (lua_State *L, int index);
```

Moves the top element into the given position (and pops it), without shifting any element (therefore replacing the value at the given position).

lua_resume lua_resume()

```
int lua_resume (lua_State *L, int narg);
```

Starts and resumes a coroutine in a given thread.

To start a coroutine, you first create a new thread (see lua_newthread()); then you push onto its stack the main function plus any arguments; then you call `lua_resume` (see lua_resume()) with `narg` being the number of arguments. This call returns when the coroutine suspends or finishes its execution. When it returns, the stack contains all values passed to `lua_yield` (see lua_yield()), or all values returned by the body function. `lua_resume` returns `LUA_YIELD` if the coroutine yields, 0 if the coroutine finishes its execution without errors, or an error code in case of errors (see lua_pcall()). In case of errors, the stack is not unwound, so you can use the debug API over it. The error message is on the top of the stack. To restart a coroutine, you put on its stack only the values to be passed as results from `lua_yield`, and then call `lua_resume`.

lua_setallocf lua_setallocf()

```
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);
```

Changes the allocator function of a given state to `f` with user data `ud`.

lua_setfenv lua_setfenv()

```
int lua_setfenv (lua_State *L, int index);
```

Pops a table from the stack and sets it as the new environment for the value at the given index. If the value at the given index is neither a function nor a thread nor a userdata, `lua_setfenv` returns 0. Otherwise it returns 1.

lua_setfield lua_setfield()

```
void lua_setfield (lua_State *L, int index, const char *k);
```

Does the equivalent to `t[k] = v`, where `t` is the value at the given valid index `index` and `v` is the value at the top of the stack.

This function pops the value from the stack. As in Lua, this function may trigger a metamethod for the "newindex" event (see lua-metatable).

lua_setglobal lua_setglobal()

```
void lua_setglobal (lua_State *L, const char *name);
```

Pops a value from the stack and sets it as the new value of global `name`. It is defined as a macro:

```
#define lua_setglobal(L,s)   lua_setfield(L, LUA_GLOBALSINDEX, s)
```

lua_setmetatable lua_setmetatable()

```
int lua_setmetatable (lua_State *L, int index);
```

Pops a table from the stack and sets it as the new metatable for the value at the given acceptable index.

lua_settable lua_settable()

```
void lua_settable (lua_State *L, int index);
```

Does the equivalent to `t[k] = v`, where `t` is the value at the given valid index `index`, `v` is the value at the top of the stack, and `k` is the value just below the top.

This function pops both the key and the value from the stack. As in Lua, this function may trigger a metamethod for the "newindex" event (see lua-metatable).

lua_settop lua_settop()

```
void lua_settop (lua_State *L, int index);
```

Accepts any acceptable index, or 0, and sets the stack top to this index. If the new top is larger than the old one, then the new elements are filled with `nil`. If `index` is 0, then all stack elements are removed.

lua_State lua_State

```
typedef struct lua_State lua_State;
```

Opaque structure that keeps the whole state of a Lua interpreter. The Lua library is fully reentrant: it has no global variables. All information about a state is kept in this structure.

A pointer to this state must be passed as the first argument to every function in the library, except to `lua_newstate` (see lua_newstate()), which creates a Lua state from scratch.

lua_status lua_status()

```
int lua_status (lua_State *L);
```

Returns the status of the thread L.

The status can be 0 for a normal thread, an error code if the thread finished its execution with an error, or `LUA_YIELD` if the thread is suspended.

lua_toboolean lua_toboolean()

```
int lua_toboolean (lua_State *L, int index);
```

Converts the Lua value at the given acceptable index to a C boolean value (0 or 1). Like all tests in Lua, `lua_toboolean` returns 1 for any Lua value different from `false` and `nil`; otherwise it returns 0. It also returns 0 when called with a non-valid index. (If you want to accept only actual boolean values, use `lua_isboolean` lua_isboolean() to test the value's type.)

lua_tocfunction lua_tocfunction()

```
lua_CFunction lua_tocfunction (lua_State *L, int index);
```

Converts a value at the given acceptable index to a C function. That value must be a C function; otherwise it returns `NULL`.

lua_tointeger lua_tointeger()

```
lua_Integer lua_tointeger (lua_State *L, int idx);
```

Converts the Lua value at the given acceptable index to the signed integral type `lua_Integer` (see lua_Integer). The Lua value must be a number or a string convertible to a number (see lua-coercion); otherwise, `lua_tointeger` returns 0.

If the number is not an integer, it is truncated in some non-specified way.

lua_tolstring lua_tolstring()

```
const char *lua_tolstring (lua_State *L, int index, size_t *len);
```

Converts the Lua value at the given acceptable index to a C string. If `len` is not `NULL`, it also sets `*len` with the string length. The Lua value must be a string or a number; otherwise, the function returns `NULL`. If the value is a number, then `lua_tolstring` also changes `the actual value in the stack to a string`. (This change confuses `lua_next` lua_next() when `lua_tolstring` is applied to keys during a table traversal.)

`lua_tolstring` returns a fully aligned pointer to a string inside the Lua state. This string always has a zero (`\0`) after its last character (as in C), but may contain other zeros in its body. Because Lua has garbage collection, there is no guarantee that the pointer returned by `lua_tolstring` will be valid after the corresponding value is removed from the stack.

lua_tonumber lua_tonumber()

```
lua_Number lua_tonumber (lua_State *L, int index);
```

Converts the Lua value at the given acceptable index to the C type `lua_Number` (see lua_Number). The Lua value must be a number or a string convertible to a number (see lua-coercion); otherwise, `lua_tonumber` returns 0.

lua_topointer lua_topointer()

```
const void *lua_topointer (lua_State *L, int index);
```

Converts the value at the given acceptable index to a generic C pointer (`void*`). The value may be a userdata, a table, a thread, or a function; otherwise, `lua_topointer` returns `NULL`. Different objects will give different pointers. There is no way to convert the pointer back to its original value.

Typically this function is used only for debug information.

lua_tostring lua_tostring()

```
const char *lua_tostring (lua_State *L, int index);
```

Equivalent to `lua_tolstring` (see lua_tolstring()) with `len` equal to `NULL`.

lua_tothread lua_tothread()

```
lua_State *lua_tothread (lua_State *L, int index);
```

Converts the value at the given acceptable index to a Lua thread (represented as `lua_State*` lua_State). This value must be a thread; otherwise, the function returns `NULL`.

lua_touserdata lua_touserdata()

```
void *lua_touserdata (lua_State *L, int index);
```

If the value at the given acceptable index is a full userdata, returns its block address. If the value is a light userdata, returns its pointer. Otherwise, it returns `NULL`.

lua_type lua_type()

```
int lua_type (lua_State *L, int index);
```

Returns the type of the value in the given acceptable index, or `LUA_TNONE` for a non-valid index (that is, an index to an "empty" stack position). The types returned by `lua_type` are coded by the following constants defined in `lua.h` : `LUA_TNIL`, `LUA_TNUMBER`, `LUA_TBOOLEAN`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, `LUA_TTHREAD`, and `LUA_TLIGHTUSERDATA`.

lua_typename lua_typename()

```
const char *lua_typename  (lua_State *L, int tp);
```

Returns the name of the type encoded by the value `tp`, which must be one the values returned by `lua_type`.

lua_Writer lua_Writer

```
typedef int (*lua_Writer) (lua_State *L,
                           const void* p,
                           size_t sz,
                           void* ud);
```

The writer function used by `lua_dump` (see lua_dump()). Every time it produces another piece of chunk, `lua_dump` calls the writer, passing along the buffer to be written (`p`), its size (`sz`), and the `data` parameter supplied to `lua_dump`.

The writer returns an error code: 0 means no errors; any other value means an error and stops `lua_dump` from calling the writer again.

lua_xmove lua_xmove()

```
void lua_xmove (lua_State *from, lua_State *to, int n);
```

Exchange values between different threads of the `same` global state.

This function pops `n` values from the stack `from`, and pushes them onto the stack `to`.

lua_yield lua_yield()

```
int lua_yield (lua_State *L, int nresults);
```

Yields a coroutine.

This function should only be called as the return expression of a C function, as follows:

```
return lua_yield (L, nresults);
```

When a C function calls `lua_yield` in that way, the running coroutine suspends its execution, and the call to `lua_resume` (see lua_resume()) that started this coroutine returns. The parameter `nresults` is the number of values from the stack that are passed as results to `lua_resume`.

lua-stackexample

As an example of stack manipulation, if the stack starts as `10 20 30 40 50*` (from bottom to top; the * marks the top), then

```
lua_pushvalue(L, 3)    --> 10 20 30 40 50 30*
lua_pushvalue(L, -1)   --> 10 20 30 40 50 30 30*
lua_remove(L, -3)      --> 10 20 30 40 30 30*
lua_remove(L,  6)      --> 10 20 30 40 30*
lua_insert(L,  1)      --> 30 10 20 30 40*
lua_insert(L, -1)      --> 30 10 20 30 40*  (no effect)
lua_replace(L, 2)      --> 30 40 20 30*
lua_settop(L, -3)      --> 30 40*
lua_settop(L,  6)      --> 30 40 nil nil nil nil*
```

3.8 The Debug Interface

lua-apiDebug

Lua has no built-in debugging facilities. Instead, it offers a special interface by means of functions and hooks. This interface allows the construction of different kinds of debuggers, profilers, and other tools that need "inside information" from the interpreter.

lua_Debug

lua_Debug

```
typedef struct lua_Debug {
    int event;
    const char *name;           /* (n) */
    const char *namewhat;       /* (n) */
    const char *what;           /* (S) */
    const char *source;         /* (S) */
    int currentline;            /* (l) */
    int nups;                   /* (u) number of upvalues */
    int linedefined;            /* (S) */
    int lastlinedefined;        /* (S) */
    char short_src[LUA_IDSIZE]; /* (S) */
    /* private part */
    other fields
} lua_Debug;
```

A structure used to carry different pieces of information about an active function. `lua_getstack` (see lua_getstack()) fills only the private part of this structure, for later use. To fill the other fields of `lua_Debug` with useful information, call `lua_getinfo` (see lua_getinfo()).

The fields of `lua_Debug` have the following meaning:

- `source` If the function was defined in a string, then `source` is that string. If the function was defined in a file, then `source` starts with a `@` followed by the file name.
- `short_src` a "printable" version of `source`, to be used in error messages.
- `linedefined` the line number where the definition of the function starts.
- `lastlinedefined` the line number where the definition of the function ends.
- `what` the string `"Lua"` if the function is a Lua function, `"C"` if it is a C function, `"main"` if it is the main part of a chunk, and `"tail"` if it was a function that did a tail call. In the latter case, Lua has no other information about the function.
- `currentline` the current line where the given function is executing. When no line information is available, `currentline` is set to -1.
- `name` a reasonable name for the given function. Because functions in Lua are first-class values, they do not have a fixed name: some functions may be the value of multiple global variables, while others may be stored only in a table field. The `lua_getinfo` function checks how the function was called to find a suitable name. If it cannot find a name, then `name` is set to `NULL`.
- `namewhat` explains the `name` field. The value of `namewhat` can be `"global"`, `"local"`, `"method"`, `"field"`, `"upvalue"`, or `""` (the empty string), according to how the function was called. (Lua uses the empty string when no other option seems to apply.) `nups` the number of upvalues of the function.

lua_gethook

lua_gethook()

```
lua_Hook lua_gethook (lua_State *L);
```

Returns the current hook function.

lua_gethookcount

lua_gethookcount()

```
int lua_gethookcount (lua_State *L);
```

Returns the current hook count.

lua_gethookmask

lua_gethookmask()

```
int lua_gethookmask (lua_State *L);
```

Returns the current hook mask.

lua_getinfo

lua_getinfo()

```
int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);
```

Returns information about a specific function or function invocation.

To get information about a function invocation, the parameter `ar` must be a valid activation record that was filled by a previous call to `lua_getstack` (see lua_getstack()) or given as argument to a hook (see lua_Hook).

To get information about a function you push it onto the stack and start the `what` string with the character `>`. (In that case, `lua_getinfo` pops the function in the top of the stack.) For instance, to know in which line a function `f` was defined, you can write the following code:

```
lua_Debug ar;
lua_getfield(L, LUA_GLOBALSINDEX, "f");  /* get global 'f' */
lua_getinfo(L, ">S", &ar);
printf("%d\n", ar.linedefined);
```

Each character in the string `what` selects some fields of the structure `ar` to be filled or a value to be pushed on the stack:

`'n'` fills in the field `name` and `namewhat` `'S'` fills in the fields `source`, `short_src`, `linedefined`, `lastlinedefined`, and `what` `'l'` fills in the field `currentline` `'u'` fills in the field `nups` `'f'` pushes onto the stack the function that is running at the given level `'L'` pushes onto the stack a table whose indices are the numbers of the lines that are valid on the function. (A `valid line` is a line with some associated code, that is, a line where you can put a break point. Non-valid lines include empty lines and comments.)

This function returns 0 on error (for instance, an invalid option in `what`).

lua_getlocal

lua_getlocal()

```
const char *lua_getlocal (lua_State *L, lua_Debug *ar, int n);
```

Gets information about a local variable of a given activation record. The parameter `ar` must be a valid activation record that was filled by a previous call to `lua_getstack` (see lua_getstack()) or given as argument to a hook (see lua_Hook). The index `n` selects which local variable to inspect (1 is the first parameter or active local variable, and so on, until the last active local variable). `lua_getlocal` pushes the variable's value onto the stack and returns its name.

Variable names starting with `(` (open parentheses) represent internal variables (loop control variables, temporaries, and C function locals).

Returns `NULL` (and pushes nothing) when the index is greater than the number of active local variables.

lua_getstack lua_getstack()

```
int lua_getstack (lua_State *L, int level, lua_Debug *ar);
```

Gets information about the interpreter runtime stack.

This function fills parts of a `lua_Debug` (see lua_Debug) structure with an identification of the `activation record` of the function executing at a given level. Level 0 is the current running function, whereas level `n+1` is the function that has called level `n`. When there are no errors, `lua_getstack` returns 1; when called with a level greater than the stack depth, it returns 0.

lua_getupvalue lua_getupvalue()

```
const char *lua_getupvalue (lua_State *L, int funcindex, int n);
```

Gets information about a closure's upvalue. (For Lua functions, upvalues are the external local variables that the function uses, and that are consequently included in its closure.) `lua_getupvalue` gets the index `n` of an upvalue, pushes the upvalue's value onto the stack, and returns its name. `funcindex` points to the closure in the stack. (Upvalues have no particular order, as they are active through the whole function. So, they are numbered in an arbitrary order.)

Returns `NULL` (and pushes nothing) when the index is greater than the number of upvalues. For C functions, this function uses the empty string `""` as a name for all upvalues.

lua_Hook lua_Hook

```
typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);
```

Type for debugging hook functions.

Whenever a hook is called, its `ar` argument has its field `event` set to the specific event that triggered the hook. Lua identifies these events with the following constants: `LUA_HOOKCALL`, `LUA_HOOKRET`, `LUA_HOOKTAILRET`, `LUA_HOOKLINE`, and `LUA_HOOKCOUNT`. Moreover, for line events, the field `currentline` is also set. To get the value of any other field in `ar`, the hook must call `lua_getinfo` (see lua_getinfo()). For return events, `event` may be `LUA_HOOKRET`, the normal value, or `LUA_HOOKTAILRET`. In the latter case, Lua is simulating a return from a function that did a tail call; in this case, it is useless to call `lua_getinfo`.

While Lua is running a hook, it disables other calls to hooks. Therefore, if a hook calls back Lua to execute a function or a chunk, this execution occurs without any calls to hooks.

lua_sethook lua_sethook()

```
int lua_sethook (lua_State *L, lua_Hook f, int mask, int count);
```

Sets the debugging hook function.

Argument `f` is the hook function. `mask` specifies on which events the hook will be called: it is formed by a bitwise `or` of the constants `LUA_MASKCALL`, `LUA_MASKRET`, `LUA_MASKLINE`, and `LUA_MASKCOUNT`. The `count` argument is only meaningful when the mask includes `LUA_MASKCOUNT`. For each event, the hook is called as explained below:

- The `call hook`: is called when the interpreter calls a function. The hook is called just after Lua enters the new function, before the function gets its arguments.
- The `return hook`: is called when the interpreter returns from a function. The hook is called just before Lua leaves the function. You have no access to the values to be returned by the function.
- The `line hook`: is called when the interpreter is about to start the execution of a new line of code, or when it jumps back in the code (even to the same line). (This event only happens while Lua is executing a Lua function.)
- The `count hook`: is called after the interpreter executes every `count` instructions. (This event only happens while Lua is executing a Lua function.)

A hook is disabled by setting `mask` to zero.

lua_setlocal lua_setlocal()

```
const char *lua_setlocal (lua_State *L, lua_Debug *ar, int n);
```

Sets the value of a local variable of a given activation record. Parameters `ar` and `n` are as in `lua_getlocal` (see lua_getlocal()). `lua_setlocal` assigns the value at the top of the stack to the variable and returns its name. It also pops the value from the stack.

Returns `NULL` (and pops nothing) when the index is greater than the number of active local variables.

lua_setupvalue lua_setupvalue()

```
const char *lua_setupvalue (lua_State *L, int funcindex, int n);
```

Sets the value of a closure's upvalue. It assigns the value at the top of the stack to the upvalue and returns its name. It also pops the value from the stack. Parameters `funcindex` and `n` are as in the `lua_getupvalue` (see lua_getupvalue()).

Returns `NULL` (and pops nothing) when the index is greater than the number of upvalues.

lua-debugexample

As an example, the following function lists the names of all local variables and upvalues for a function at a given level of the stack:

```
int listvars (lua_State *L, int level) {
  lua_Debug ar;
  int i;
  const char *name;
  if (lua_getstack(L, level, &ar) == 0)
    return 0;  /* failure: no such level in the stack */
  i = 1;
  while ((name = lua_getlocal(L, &ar, i++)) != NULL) {
    printf("local %d %s\n", i-1, name);
    lua_pop(L, 1);  /* remove variable value */
  }
  lua_getinfo(L, "f", &ar);  /* retrieves function */
  i = 1;
  while ((name = lua_getupvalue(L, -1, i++)) != NULL) {
    printf("upvalue %d %s\n", i-1, name);
    lua_pop(L, 1);  /* remove upvalue value */
  }
  return 1;
}
```

4 THE AUXILIARY LIBRARY

lua-aux

The auxiliary library provides several convenient functions to interface C with Lua. While the basic API provides the primitive functions for all interactions between C and Lua, the auxiliary library provides higher-level functions for some common tasks.

All functions from the auxiliary library are defined in header file `lauxlib.h` and have a prefix `luaL_`.

All functions in the auxiliary library are built on top of the basic API, and so they provide nothing that cannot be done with this API.

Several functions in the auxiliary library are used to check C function arguments. Their names are always `luaL_check*` or `luaL_opt*`. All of these functions raise an error if the check is not satisfied. Because the error message is formatted for arguments (e.g., "bad argument #1"), you should not use these functions for other stack values.

4.1 Functions and Types

lua-auxFunctions

Here we list all functions and types from the auxiliary library in alphabetical order.

luaL_addchar

luaL_addchar()

```
void luaL_addchar (luaL_Buffer *B, char c);
```

Adds the character `c` to the buffer `B` (see luaL_Buffer).

luaL_addlstring

luaL_addlstring()

```
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
```

Adds the string pointed to by `s` with length `l` to the buffer `B` (see luaL_Buffer). The string may contain embedded zeros.

luaL_addsize

luaL_addsize()

```
void luaL_addsize (luaL_Buffer *B, size_t n);
```

Adds to the buffer `B` (see luaL_Buffer) a string of length `n` previously copied to the buffer area (see luaL_prepbuffer()).

luaL_addstring

luaL_addstring()

```
void luaL_addstring (luaL_Buffer *B, const char *s);
```

Adds the zero-terminated string pointed to by `s` to the buffer `B` (see luaL_Buffer). The string may not contain embedded zeros.

luaL_addvalue

luaL_addvalue()

```
void luaL_addvalue (luaL_Buffer *B);
```

Adds the value at the top of the stack to the buffer `B` (see luaL_Buffer). Pops the value.

This is the only function on string buffers that can (and must) be called with an extra element on the stack, which is the value to be added to the buffer.

luaL_argcheck

luaL_argcheck()

```
void luaL_argcheck (lua_State *L,
                    int cond,
                    int narg,
                    const char *extramsg);
```

Checks whether `cond` is true. If not, raises an error with the following message, where `func` is retrieved from the call stack:

```
bad argument #<narg> to <func> (<extramsg>)
```

luaL_argerror

luaL_argerror()

```
int luaL_argerror (lua_State *L, int narg, const char *extramsg);
```

Raises an error with the following message, where `func` is retrieved from the call stack:

```
bad argument #<narg> to <func> (<extramsg>)
```

This function never returns, but it is an idiom to use it in C functions as `return luaL_argerror( args )`.

luaL_Buffer luaL_Buffer

```
typedef struct luaL_Buffer luaL_Buffer;
```

Type for a `string buffer`.

A string buffer allows C code to build Lua strings piecemeal. Its pattern of use is as follows:

- First you declare a variable b of type `luaL_Buffer`.
- Then you initialize it with a call `luaL_buffinit(L, &b)` (see luaL_buffinit()).
- Then you add string pieces to the buffer calling any of the `luaL_add*` functions.
- You finish by calling `luaL_pushresult(&b)` (see luaL_pushresult()). This call leaves the final string on the top of the stack.

During its normal operation, a string buffer uses a variable number of stack slots. So, while using a buffer, you cannot assume that you know where the top of the stack is. You can use the stack between successive calls to buffer operations as long as that use is balanced; that is, when you call a buffer operation, the stack is at the same level it was immediately after the previous buffer operation. (The only exception to this rule is `luaL_addvalue` luaL_addvalue().) After calling `luaL_pushresult` the stack is back to its level when the buffer was initialized, plus the final string on its top.

luaL_buffinit luaL_buffinit()

```
void luaL_buffinit (lua_State *L, luaL_Buffer *B);
```

Initializes a buffer B. This function does not allocate any space; the buffer must be declared as a variable (see luaL_Buffer).

luaL_callmeta luaL_callmeta()

```
int luaL_callmeta (lua_State *L, int obj, const char *e);
```

Calls a metamethod.

If the object at index `obj` has a metatable and this metatable has a field `e`, this function calls this field and passes the object as its only argument. In this case this function returns 1 and pushes onto the stack the value returned by the call. If there is no metatable or no metamethod, this function returns 0 (without pushing any value on the stack).

luaL_checkany luaL_checkany()

```
void luaL_checkany (lua_State *L, int narg);
```

Checks whether the function has an argument of any type (including `nil`) at position `narg`.

luaL_checkint luaL_checkint()

```
int luaL_checkint (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a number and returns this number cast to an `int`.

luaL_checkinteger luaL_checkinteger()

```
lua_Integer luaL_checkinteger (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a number and returns this number cast to a `lua_Integer` (see lua_Integer).

luaL_checklong luaL_checklong()

```
long luaL_checklong (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a number and returns this number cast to a `long`.

luaL_checklstring luaL_checklstring()

```
const char *luaL_checklstring (lua_State *L, int narg, size_t *l);
```

Checks whether the function argument `narg` is a string and returns this string; if `l` is not `NULL` fills `*l` with the string's length.

luaL_checknumber luaL_checknumber()

```
lua_Number luaL_checknumber (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a number and returns this number (see lua_Number).

luaL_checkoption luaL_checkoption()

```
int luaL_checkoption (lua_State *L,
                      int narg,
                      const char *def,
                      const char *const lst[]);
```

Checks whether the function argument `narg` is a string and searches for this string in the array `lst` (which must be NULL-terminated). Returns the index in the array where the string was found. Raises an error if the argument is not a string or if the string cannot be found.

If `def` is not `NULL`, the function uses `def` as a default value when there is no argument `narg` or if this argument is `nil`.

This is a useful function for mapping strings to C enums. (The usual convention in Lua libraries is to use strings instead of numbers to select options.)

luaL_checkstack luaL_checkstack()

```
void luaL_checkstack (lua_State *L, int sz, const char *msg);
```

Grows the stack size to `top + sz` elements, raising an error if the stack cannot grow to that size. `msg` is an additional text to go into the error message.

luaL_checkstring luaL_checkstring()

```
const char *luaL_checkstring (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a string and returns this string.

luaL_checktype luaL_checktype()

```
void luaL_checktype (lua_State *L, int narg, int t);
```

Checks whether the function argument `narg` has type `t` (see lua_type()).

luaL_checkudata luaL_checkudata()

```
void *luaL_checkudata (lua_State *L, int narg, const char *tname);
```

Checks whether the function argument `narg` is a userdata of the type `tname` (see luaL_newmetatable()).

luaL_dofile luaL_dofile()

```
int luaL_dofile (lua_State *L, const char *filename);
```

Loads and runs the given file. It is defined as the following macro:

```
(luaL_loadfile(L, filename) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

It returns 0 if there are no errors or 1 in case of errors.

luaL_dostring luaL_dostring()

```
int luaL_dostring (lua_State *L, const char *str);
```

Loads and runs the given string. It is defined as the following macro:

```
(luaL_loadstring(L, str) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

It returns 0 if there are no errors or 1 in case of errors.

luaL_error luaL_error()

```
int luaL_error (lua_State *L, const char *fmt, ...);
```

Raises an error. The error message format is given by `fmt` plus any extra arguments, following the same rules of `lua_pushfstring` (see lua_pushfstring()). It also adds at the beginning of the message the file name and the line number where the error occurred, if this information is available.

This function never returns, but it is an idiom to use it in C functions as `return luaL_error( args )`.

luaL_getmetafield luaL_getmetafield()

```
int luaL_getmetafield (lua_State *L, int obj, const char *e);
```

Pushes onto the stack the field `e` from the metatable of the object at index `obj`. If the object does not have a metatable, or if the metatable does not have this field, returns 0 and pushes nothing.

luaL_getmetatable luaL_getmetatable()

```
void luaL_getmetatable (lua_State *L, const char *tname);
```

Pushes onto the stack the metatable associated with name `tname` in the registry (see luaL_newmetatable()).

luaL_gsub luaL_gsub()

```
const char *luaL_gsub (lua_State *L,
                       const char *s,
                       const char *p,
                       const char *r);
```

Creates a copy of string `s` by replacing any occurrence of the string `p` with the string `r`. Pushes the resulting string on the stack and returns it.

luaL_loadbuffer luaL_loadbuffer()

```
int luaL_loadbuffer (lua_State *L,
                     const char *buff,
                     size_t sz,
                     const char *name);
```

Loads a buffer as a Lua chunk. This function uses `lua_load` (see lua_load()) to load the chunk in the buffer pointed to by `buff` with size `sz`.

This function returns the same results as `lua_load`. `name` is the chunk name, used for debug information and error messages.

luaL_loadfile luaL_loadfile()

```
int luaL_loadfile (lua_State *L, const char *filename);
```

Loads a file as a Lua chunk. This function uses `lua_load` (see lua_load()) to load the chunk in the file named `filename`. If `filename` is `NULL`, then it loads from the standard input. The first line in the file is ignored if it starts with a #.

This function returns the same results as `lua_load`, but it has an extra error code `LUA_ERRFILE` if it cannot open/read the file.

As `lua_load`, this function only loads the chunk; it does not run it.

luaL_loadstring luaL_loadstring()

```
int luaL_loadstring (lua_State *L, const char *s);
```

Loads a string as a Lua chunk. This function uses `lua_load` (see lua_load()) to load the chunk in the zero-terminated string `s`.

This function returns the same results as `lua_load`.

Also as `lua_load`, this function only loads the chunk; it does not run it.

luaL_newmetatable luaL_newmetatable()

```
int luaL_newmetatable (lua_State *L, const char *tname);
```

If the registry already has the key `tname`, returns 0. Otherwise, creates a new table to be used as a metatable for userdata, adds it to the registry with key `tname`, and returns 1.

In both cases pushes onto the stack the final value associated with `tname` in the registry.

luaL_newstate luaL_newstate()

```
lua_State *luaL_newstate (void);
```

Creates a new Lua state. It calls `lua_newstate` (see lua_newstate()) with an allocator based on the standard C `realloc` function and then sets a panic function (see lua_atpanic()) that prints an error message to the standard error output in case of fatal errors.

Returns the new state, or `NULL` if there is a memory allocation error.

luaL_openlibs luaL_openlibs()

```
void luaL_openlibs (lua_State *L);
```

Opens all standard Lua libraries into the given state. See also lua-openlibs for details on how to open individual libraries.

luaL_optint luaL_optint()

```
int luaL_optint (lua_State *L, int narg, int d);
```

If the function argument `narg` is a number, returns this number cast to an `int`. If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

luaL_optinteger luaL_optinteger()

```
lua_Integer luaL_optinteger (lua_State *L,
                             int narg,
                             lua_Integer d);
```

If the function argument `narg` is a number, returns this number cast to a `lua_Integer` (see lua_Integer). If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

luaL_optlong luaL_optlong()

```
long luaL_optlong (lua_State *L, int narg, long d);
```

If the function argument `narg` is a number, returns this number cast to a `long`. If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

luaL_optlstring luaL_optlstring()

```
const char *luaL_optlstring (lua_State *L,
                             int narg,
                             const char *d,
                             size_t *l);
```

If the function argument `narg` is a string, returns this string. If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

If `l` is not `NULL`, fills the position `*l` with the results' length.

luaL_optnumber luaL_optnumber()

```
lua_Number luaL_optnumber (lua_State *L, int narg, lua_Number d);
```

If the function argument `narg` is a number, returns this number. If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

luaL_optstring luaL_optstring()

```
const char *luaL_optstring (lua_State *L,
                            int narg,
                            const char *d);
```

If the function argument `narg` is a string, returns this string. If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

luaL_prepbuffer luaL_prepbuffer()

```
char *luaL_prepbuffer (luaL_Buffer *B);
```

Returns an address to a space of size `LUAL_BUFFERSIZE` where you can copy a string to be added to buffer `B` (see luaL_Buffer). After copying the string into this space you must call `luaL_addsize` (see luaL_addsize()) with the size of the string to actually add it to the buffer.

luaL_pushresult luaL_pushresult()

```
void luaL_pushresult (luaL_Buffer *B);
```

Finishes the use of buffer B leaving the final string on the top of the stack.

luaL_ref luaL_ref()

```
int luaL_ref (lua_State *L, int t);
```

Creates and returns a `reference`, in the table at index `t`, for the object at the top of the stack (and pops the object).

A reference is a unique integer key. As long as you do not manually add integer keys into table `t`, `luaL_ref` ensures the uniqueness of the key it returns. You can retrieve an object referred by reference `r` by calling `lua_rawgeti(L, t, r)` (see lua_rawgeti()). Function `luaL_unref` (see luaL_unref()) frees a reference and its associated object.

If the object at the top of the stack is `nil`, `luaL_ref` returns the constant `LUA_REFNIL`. The constant `LUA_NOREF` is guaranteed to be different from any reference returned by `luaL_ref`.

luaL_Reg luaL_Reg

```
typedef struct luaL_Reg {
    const char *name;
    lua_CFunction func;
} luaL_Reg;
```

Type for arrays of functions to be registered by `luaL_register` (see luaL_register()). `name` is the function name and `func` is a pointer to the function. Any array of `luaL_Reg` must end with a sentinel entry in which both `name` and `func` are `NULL`.

luaL_register luaL_register()

```
void luaL_register (lua_State *L,
                    const char *libname,
                    const luaL_Reg *l);
```

Opens a library.

When called with `libname` equal to `NULL`, it simply registers all functions in the list `l` (see luaL_Reg) into the table on the top of the stack.

When called with a non-null `libname`, `luaL_register` creates a new table `t`, sets it as the value of the global variable `libname`, sets it as the value of `package.loaded[libname]`, and registers on it all functions in the list `l`. If there is a table in `package.loaded[libname]` or in variable `libname`, reuses this table instead of creating a new one.

In any case the function leaves the table on the top of the stack.

luaL_typename luaL_typename()

```
const char *luaL_typename (lua_State *L, int idx);
```

Returns the name of the type of the value at index `idx`.

luaL_typerror luaL_typerror()

```
int luaL_typerror (lua_State *L, int narg, const char *tname);
```

Generates an error with a message like the following:

`location : bad argument narg to 'func' ( tname expected, got rt )`

where `location` is produced by `luaL_where` (see luaL_where()), `func` is the name of the current function, and `rt` is the type name of the actual argument.

luaL_unref luaL_unref()

```
void luaL_unref (lua_State *L, int t, int ref);
```

Releases reference `ref` from the table at index `t` (see luaL_ref()). The entry is removed from the table, so that the referred object can be collected. The reference `ref` is also freed to be used again.

If `ref` is `LUA_NOREF` or `LUA_REFNIL`, `luaL_unref` does nothing.

luaL_where luaL_where()

```
void luaL_where (lua_State *L, int lvl);
```

Pushes onto the stack a string identifying the current position of the control at level `lvl` in the call stack. Typically this string has the following format:

`chunkname:currentline:`

Level 0 is the running function, level 1 is the function that called the running function, etc.

This function is used to build a prefix for error messages.

5 STANDARD LIBRARIES lua-lib

The standard libraries provide useful functions that are implemented directly through the C API. Some of these functions provide essential services to the language (e.g., `type` and `getmetatable`); others provide access to "outside" services (e.g., I/O); and others could be implemented in Lua itself, but are quite useful or have critical performance requirements that deserve an implementation in C (e.g., `sort`).

All libraries are implemented through the official C API and are provided as separate C modules. Currently, Lua has the following standard libraries:

- basic library;
- package library;
- string manipulation;
- table manipulation;
- mathematical functions (sin, log, etc.);
- input and output;
- operating system facilities;
- debug facilities.

Except for the basic and package libraries, each library provides all its functions as fields of a global table or as methods of its objects.

lua-openlibs

To have access to these libraries, the C host program should call the `luaL_openlibs` function, which opens all standard libraries (see luaL_openlibs()). Alternatively, the host program can open the libraries individually by calling `luaopen_base` (for the basic library), `luaopen_package` (for the package library), `luaopen_string` (for the string library), `luaopen_table` (for the table library), `luaopen_math` (for the mathematical library), `luaopen_io` (for the I/O and the Operating System libraries), and `luaopen_debug` (for the debug library). These functions are declared in `lualib.h` and should not be called directly: you must call them like any other Lua C function, e.g., by using `lua_call` (see lua_call()).

5.1 Basic Functions

lua-lib-core

The basic library provides some core functions to Lua. If you do not include this library in your application, you should check carefully whether you need to provide implementations for some of its facilities.

assert({v} [, {message}])
assert()
Issues an error when the value of its argument v is false (i.e., `nil` or `false`); otherwise, returns all its arguments. `message` is an error message; when absent, it defaults to "assertion failed!"

collectgarbage({opt} [, {arg}])
collectgarbage()
This function is a generic interface to the garbage collector. It performs different functions according to its first argument, {opt}:

`"stop"` stops the garbage collector. `"restart"` restarts the garbage collector. `"collect"` performs a full garbage-collection cycle. `"count"` returns the total memory in use by Lua (in Kbytes). `"step"` performs a garbage-collection step. The step "size" is controlled by {arg} (larger values mean more steps) in a non-specified way. If you want to control the step size you must experimentally tune the value of {arg}. Returns `true` if the step finished a collection cycle. `"setpause"` sets {arg} /100 as the new value for the `pause` of the collector (see lua-gc). `"setstepmul"` sets {arg} /100 as the new value for the `step multiplier` of the collector (see lua-gc).

dofile({filename})
dofile()
Opens the named file and executes its contents as a Lua chunk. When called without arguments, `dofile` executes the contents of the standard input (`stdin`). Returns all values returned by the chunk. In case of errors, `dofile` propagates the error to its caller (that is, `dofile` does not run in protected mode).

error({message} [, {level}])
error()
Terminates the last protected function called and returns `message` as the error message. Function {error} never returns.

Usually, {error} adds some information about the error position at the beginning of the message. The {level} argument specifies how to get the error position. With level 1 (the default), the error position is where the {error} function was called. Level 2 points the error to where the function that called {error} was called; and so on. Passing a level 0 avoids the addition of error position information to the message.

_G
_G
A global variable (not a function) that holds the global environment (that is, `_G._G = _G`). Lua itself does not use this variable; changing its value does not affect any environment, nor vice-versa. (Use `setfenv` to change environments.)

getfenv({f})
getfenv()
Returns the current environment in use by the function. {f} can be a Lua function or a number that specifies the function at that stack level: Level 1 is the function calling `getfenv`. If the given function is not a Lua function, or if {f} is 0, `getfenv` returns the global environment. The default for {f} is 1.

getmetatable({object})
getmetatable()
If {object} does not have a metatable, returns `nil`. Otherwise, if the object's metatable has a `"__metatable"` field, returns the associated value. Otherwise, returns the metatable of the given object.

ipairs({t})
ipairs()
Returns three values: an iterator function, the table {t}, and 0, so that the construction

```
for i,v in ipairs(t) do body end
```

will iterate over the pairs `(1,t[1])`, `(2,t[2])`, ..., up to the first integer key absent from the table.

load({func} [, {chunkname}])
load()
Loads a chunk using function {func} to get its pieces. Each call to {func} must return a string that concatenates with previous results. A return of `nil` (or no value) signals the end of the chunk.

If there are no errors, returns the compiled chunk as a function; otherwise, returns `nil` plus the error message. The environment of the returned function is the global environment.

{chunkname} is used as the chunk name for error messages and debug information.

loadfile([{filename}])
loadfile()
Similar to `load` (see load()), but gets the chunk from file {filename} or from the standard input, if no file name is given.

loadstring({string} [, {chunkname}])
loadstring()
Similar to `load` (see load()), but gets the chunk from the given {string}.

To load and run a given string, use the idiom

```
assert(loadstring(s))()
```

next({table} [, {index}]) next()
Allows a program to traverse all fields of a table. Its first argument is a table and its second argument is an index in this table. next returns the next index of the table and its associated value. When called with nil as its second argument, next returns an initial index and its associated value. When called with the last index, or with nil in an empty table, next returns nil. If the second argument is absent, then it is interpreted as nil. In particular, you can use next(t) to check whether a table is empty.

The order in which the indices are enumerated is not specified, even for numeric indices. (To traverse a table in numeric order, use a numerical for or the ipairs() function.)

The behavior of next is undefined if, during the traversal, you assign any value to a non-existent field in the table. You may however modify existing fields. In particular, you may clear existing fields.

pairs({t}) pairs()
Returns three values: the next() function, the table {t}, and nil, so that the construction

```
for k,v in pairs(t) do body end
```

will iterate over all key-value pairs of table {t}.

pcall({f}, {arg1}, {...}) pcall()
Calls function {f} with the given arguments in protected mode. This means that any error inside {f} is not propagated; instead, pcall catches the error and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In such case, pcall also returns all results from the call, after this first result. In case of any error, pcall returns false plus the error message.

print({...}) print()
Receives any number of arguments, and prints their values to stdout, using the tostring tostring() function to convert them to strings. print is not intended for formatted output, but only as a quick way to show a value, typically for debugging. For formatted output, use string.format (see string.format()).

rawequal({v1}, {v2}) rawequal()
Checks whether {v1} is equal to {v2}, without invoking any metamethod. Returns a boolean.

rawget({table}, {index}) rawget()
Gets the real value of table[index], without invoking any metamethod. {table} must be a table; {index} may be any value.

rawset({table}, {index}, {value}) rawset()
Sets the real value of table[index] to {value}, without invoking any metamethod. {table} must be a table, {index} any value different from nil, and {value} any Lua value.

This function returns {table}.

select({index}, {...}) select()
If {index} is a number, returns all arguments after argument number {index}. Otherwise, {index} must be the string "#", and select returns the total number of extra arguments it received.

setfenv({f}, {table}) setfenv()
Sets the environment to be used by the given function. {f} can be a Lua function or a number that specifies the function at that stack level: Level 1 is the function calling setfenv. setfenv returns the given function.

As a special case, when {f} is 0 setfenv changes the environment of the running thread. In this case, setfenv returns no values.

setmetatable({table}, {metatable}) setmetatable()
Sets the metatable for the given table. (You cannot change the metatable of other types from Lua, only from C.) If {metatable} is nil, removes the metatable of the given table. If the original metatable has a "__metatable" field, raises an error.

This function returns {table}.

tonumber({e} [, {base}]) tonumber()
Tries to convert its argument to a number. If the argument is already a number or a string convertible to a number, then tonumber returns this number; otherwise, it returns nil.

An optional argument specifies the base to interpret the numeral. The base may be any integer between 2 and 36, inclusive. In bases above 10, the letter A (in either upper or lower case) represents 10, B represents 11, and so forth, with Z' representing 35. In base 10 (the default), the number may have a decimal part, as well as an optional exponent part (see lua-lexical). In other bases, only unsigned integers are accepted.

tostring({e}) tostring()
Receives an argument of any type and converts it to a string in a reasonable format. For complete control of how numbers are converted, use string.format (see string.format()).

__tostring
If the metatable of {e} has a "__tostring" field, tostring calls the corresponding value with {e} as argument, and uses the result of the call as its result.

type({v}) lua-type()
Returns the type of its only argument, coded as a string. The possible results of this function are "nil" (a string, not the value nil), "number", "string", "boolean", "table", "function", "thread", and "userdata".

unpack({list} [, {i} [, {j}]]) unpack()
Returns the elements from the given table. This function is equivalent to

```
return list[i], list[i+1], ..., list[j]
```

except that the above code can be written only for a fixed number of elements. By default, {i} is 1 and {j} is the length of the list, as defined by the length operator (see lua-length).

_VERSION _VERSION
A global variable (not a function) that holds a string containing the current interpreter version. The current contents of this string is "Lua 5.1"

.

xpcall({f}, {err}) xpcall()
This function is similar to pcall (see pcall()), except that you can set a new error handler.

xpcall calls function {f} in protected mode, using {err} as the error handler. Any error inside {f} is not propagated; instead, xpcall catches the error, calls the {err} function with the original error object, and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In this case, xpcall also returns all results from the call, after this first result. In case of any error, xpcall returns false plus the result from {err}.

5.2 Coroutine Manipulation lua-lib-coroutine

The operations related to coroutines comprise a sub-library of the basic library and come inside the table coroutine. See lua-coroutine for a general description of coroutines.

coroutine.create({f}) coroutine.create()
Creates a new coroutine, with body {f}. {f} must be a Lua function. Returns this new coroutine, an object with type "thread".

coroutine.resume({co} [, {val1}, {...}]) coroutine.resume()
Starts or continues the execution of coroutine {co}. The first time you resume a coroutine, it starts running its body. The values {val1}, {...} are passed as arguments to the body function. If the coroutine has yielded, resume restarts it; the values {val1}, {...} are passed as the results from the yield.

If the coroutine runs without any errors, resume returns true plus any values passed to yield (if the coroutine yields) or any values returned by the body function(if the coroutine terminates). If there is any error, resume returns false plus the error message.

coroutine.running() coroutine.running()
Returns the running coroutine, or nil when called by the main thread.

coroutine.status({co}) coroutine.status()
Returns the status of coroutine {co}, as a string: "running", if the coroutine is running (that is, it called status); "suspended", if the coroutine is suspended in a call to yield, or if it has not started running yet; "normal" if the coroutine is active but not running (that is, it has resumed another coroutine); and "dead" if the coroutine has finished its body function, or if it has stopped with an error.

coroutine.wrap({f}) coroutine.wrap()
Creates a new coroutine, with body {f}. {f} must be a Lua function. Returns a function that resumes the coroutine each time it is called. Any arguments passed to the function behave as the extra arguments to resume. Returns the same values returned by resume, except the first boolean. In case of error, propagates the error.

coroutine.yield({...}) coroutine.yield()
Suspends the execution of the calling coroutine. The coroutine cannot be running a C function, a metamethod, or an iterator. Any arguments to yield are passed as extra results to resume.

5.3 Modules lua-modules

The package library provides basic facilities for loading and building modules in Lua. It exports two of its functions directly in the global environment: require and module (see require() and module()). Everything else is exported in a table package.

module({name} [, {...}]) module()
Creates a module. If there is a table in package.loaded[name], this table is the module. Otherwise, if there is a global table t with the given name, this table is the module. Otherwise creates a new table t and sets it as the value of the global {name} and the value of package.loaded[name]. This function also initializes t._NAME with the given name, t._M with the module (t itself), and t._PACKAGE with the package name (the full module name minus last component; see below). Finally, module sets t as the new environment of the current function and the new value of package.loaded[name], so that require() returns t.

If {name} is a compound name (that is, one with components separated by dots), module creates (or reuses, if they already exist) tables for each component. For instance, if {name} is a.b.c, then module stores the module table in field c of field b of global a.

This function may receive optional options after the module name, where each option is a function to be applied over the module.

require({modname}) require()
Loads the given module. The function starts by looking into the package.loaded table to determine whether {modname} is already loaded. If it is, then require returns the value stored at package.loaded[modname]. Otherwise, it tries to find a loader for the module.

To find a loader, first require queries package.preload[modname]. If it has a value, this value (which should be a function) is the loader. Otherwise require searches for a Lua loader using the path stored in package.path. If that also fails, it searches for a C loader using the path stored in package.cpath. If that also fails, it tries an all-in-one loader (see below).

When loading a C library, require first uses a dynamic link facility to link the application with the library. Then it tries to find a C function inside this library to be used as the loader. The name of this C function is the string "luaopen_" concatenated with a copy of the module name where each dot is replaced by an underscore. Moreover, if the module name has a hyphen, its prefix up to (and including) the first hyphen is removed. For instance, if the module name is a.v1-b.c, the function name will be luaopen_b_c.

If require finds neither a Lua library nor a C library for a module, it calls the all-in-one loader. This loader searches the C path for a library for the root name of the given module. For instance, when requiring a.b.c, it will search for a C library for a. If found, it looks into it for an open function for the submodule; in our example, that would be luaopen_a_b_c. With this facility, a package can pack several C submodules into one single library, with each submodule keeping its original open function.

Once a loader is found, require calls the loader with a single argument, {modname}. If the loader returns any value, require assigns the returned value to package.loaded[modname]. If the loader returns no value and has not assigned any value to package.loaded[modname], then require assigns true to this entry. In any case, require returns the final value of package.loaded[modname].

If there is any error loading or running the module, or if it cannot find any loader for the module, then require signals an error.

package.cpath package.cpath
The path used by require to search for a C loader.

Lua initializes the C path `package.cpath` in the same way it initializes the Lua path `package.path`, using the environment variable `LUA_CPATH` (plus another default path defined in `luaconf.h`).

package.loaded package.loaded
A table used by `require` to control which modules are already loaded. When you require a module `modname` and `package.loaded[modname]` is not false, `require` simply returns the value stored there.

package.loadlib({libname}, {funcname}) package.loadlib()
Dynamically links the host program with the C library {libname}. Inside this library, looks for a function {funcname} and returns this function as a C function. (So, {funcname} must follow the protocol (see lua_CFunction)).

This is a low-level function. It completely bypasses the package and module system. Unlike `require`, it does not perform any path searching and does not automatically adds extensions. {libname} must be the complete file name of the C library, including if necessary a path and extension. {funcname} must be the exact name exported by the C library (which may depend on the C compiler and linker used).

This function is not supported by ANSI C. As such, it is only available on some platforms (Windows, Linux, Mac OS X, Solaris, BSD, plus other Unix systems that support the `dlfcn` standard).

package.path package.path
The path used by `require` to search for a Lua loader.

At start-up, Lua initializes this variable with the value of the environment variable `LUA_PATH` or with a default path defined in `luaconf.h`, if the environment variable is not defined. Any `";;"` in the value of the environment variable is replaced by the default path.

A path is a sequence of `templates` separated by semicolons. For each template, `require` will change each interrogation mark in the template by `filename`, which is `modname` with each dot replaced by a "directory separator" (such as `"/"` in Unix); then it will try to load the resulting file name. So, for instance, if the Lua path is

```
"./?.lua;./?.lc;/usr/local/?/init.lua"
```

the search for a Lua loader for module `foo` will try to load the files `./foo.lua`, `./foo.lc`, and `/usr/local/foo/init.lua`, in that order.

package.preload package.preload
A table to store loaders for specific modules (see require()).

package.seeall({module}) package.seeall()
Sets a metatable for {module} with its `__index` field referring to the global environment, so that this module inherits values from the global environment. To be used as an option to function {module}.

5.4 String Manipulation lua-lib-string

This library provides generic functions for string manipulation, such as finding and extracting substrings, and pattern matching. When indexing a string in Lua, the first character is at position 1 (not at 0, as in C). Indices are allowed to be negative and are interpreted as indexing backwards, from the end of the string. Thus, the last character is at position -1, and so on.

The string library provides all its functions inside the table `string`. It also sets a metatable for strings where the `__index` field points to the `string` table. Therefore, you can use the string functions in object-oriented style. For instance, `string.byte(s, i)` can be written as `s:byte(i)`.

string.byte({s} [, {i} [, {j}]]) string.byte()
Returns the internal numerical codes of the characters `s[i]`, `s[i+1]`,..., `s[j]`. The default value for {i} is 1; the default value for {j} is {i}.

Note that numerical codes are not necessarily portable across platforms.

string.char({...}) string.char()
Receives zero or more integers. Returns a string with length equal to the number of arguments, in which each character has the internal numerical code equal to its correspondent argument.

Note that numerical codes are not necessarily portable across platforms.

string.dump({function}) string.dump()
Returns a string containing a binary representation of the given function, so that a later loadstring() on this string returns a copy of the function. {function} must be a Lua function without upvalues.

string.find({s}, {pattern} [, {init} [, {plain}]]) string.find()
Looks for the first match of {pattern} in the string {s}. If it finds a match, then {find} returns the indices of {s} where this occurrence starts and ends; otherwise, it returns `nil`. A third, optional numerical argument {init} specifies where to start the search; its default value is 1 and may be negative. A value of {true} as a fourth, optional argument {plain} turns off the pattern matching facilities, so the function does a plain "find substring" operation, with no characters in {pattern} being considered "magic". Note that if {plain} is given, then {init} must be given as well.

If the pattern has captures, then in a successful match the captured values are also returned, after the two indices.

string.format({formatstring}, {...}) string.format()
Returns a formatted version of its variable number of arguments following the description given in its first argument (which must be a string). The format string follows the same rules as the `printf` family of standard C functions. The only differences are that the options/modifiers `*`, `l`, `L`, `n`, `p`, and `h` are not supported and that there is an extra option, `q`. The `q` option formats a string in a form suitable to be safely read back by the Lua interpreter: the string is written between double quotes, and all double quotes, newlines, embedded zeros, and backslashes in the string are correctly escaped when written. For instance, the call

```
string.format('%q', 'a string with "quotes" and \n new line')
```

will produce the string:

```
"a string with \"quotes\" and \
 new line"
```

The options `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X`, and `x` all expect a number as argument, whereas `q` and `s` expect a string.

This function does not accept string values containing embedded zeros.

string.gmatch({s}, {pattern}) string.gmatch()
Returns an iterator function that, each time it is called, returns the next captures from {pattern} over string {s}.

If {pattern} specifies no captures, then the whole match is produced in each call.

As an example, the following loop

```
s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
  print(w)
end
```

will iterate over all the words from string {s}, printing one per line. The next example collects all pairs key=value from the given string into a table:

```
t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s, "(%w+)=(%w+)") do
  t[k] = v
end
```

string.gsub({s}, {pattern}, {repl} [, {n}]) string.gsub()
Returns a copy of {s} in which all occurrences of the {pattern} have been replaced by a replacement string specified by {repl}, which may be a string, a table, or a function. gsub also returns, as its second value, the total number of substitutions made.

If {repl} is a string, then its value is used for replacement. The character % works as an escape character: any sequence in {repl} of the form %n, with {n} between 1 and 9, stands for the value of the {n} -th captured substring (see below). The sequence %0 stands for the whole match. The sequence %% stands for a single %.

If {repl} is a table, then the table is queried for every match, using the first capture as the key; if the pattern specifies no captures, then the whole match is used as the key.

If {repl} is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order; if the pattern specifies no captures, then the whole match is passed as a sole argument.

If the value returned by the table query or by the function call is a string or a number, then it is used as the replacement string; otherwise, if it is false or nil, then there is no replacement (that is, the original match is kept in the string).

The optional last parameter {n} limits the maximum number of substitutions to occur. For instance, when {n} is 1 only the first occurrence of pattern is replaced.

Here are some examples:

```
x = string.gsub("hello world", "(%w+)", "%1 %1")
--> x="hello hello world world"

x = string.gsub("hello world", "%w+", "%0 %0", 1)
--> x="hello hello world"

x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
--> x="world hello Lua from"

x = string.gsub("home = $HOME, user = $USER", "%$(%w+)", os.getenv)
--> x="home = /home/roberto, user = roberto"

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
      return loadstring(s)()
    end)
--> x="4+5 = 9"

local t = {name="lua", version="5.1"}
x = string.gsub("$name%-$version.tar.gz", "%$(%w+)", t)
--> x="lua-5.1.tar.gz"
```

string.len({s}) string.len()
Receives a string and returns its length. The empty string "" has length 0. Embedded zeros are counted, so "a\000b\000c" has length 5.

string.lower({s}) string.lower()
Receives a string and returns a copy of this string with all uppercase letters changed to lowercase. All other characters are left unchanged. The definition of what an uppercase letter is depends on the current locale.

string.match({s}, {pattern} [, {init}]) string.match()
Looks for the first match of {pattern} in the string {s}. If it finds one, then match returns the captures from the pattern; otherwise it returns nil. If {pattern} specifies no captures, then the whole match is returned. A third, optional numerical argument {init} specifies where to start the search; its default value is 1 and may be negative.

string.rep({s}, {n}) string.rep()
Returns a string that is the concatenation of {n} copies of the string {s}.

string.reverse({s}) string.reverse()
Returns a string that is the string {s} reversed.

string.sub({s}, {i} [, {j}]) string.sub()
Returns the substring of {s} that starts at {i} and continues until {j}; {i} and {j} may be negative. If {j} is absent, then it is assumed to be equal to -1 (which is the same as the string length). In particular, the call string.sub(s,1,j) returns a prefix of {s} with length {j}, and string.sub(s,-i) returns a suffix of {s} with length {i}.

string.upper({s}) string.upper()
Receives a string and returns a copy of that string with all lowercase letters changed to uppercase. All other characters are left unchanged. The definition of what a lowercase letter is depends on the current locale.

5.4.1 Patterns lua-pattern

A character class is used to represent a set of characters. The following combinations are allowed in describing a character class:

- x (where x is not one of the magic characters `^$()%.[]*+-?`) represents the character x itself.
- `.` (a dot) represents all characters.
- `%a` represents all letters.
- `%c` represents all control characters.
- `%d` represents all digits.
- `%l` represents all lowercase letters.
- `%p` represents all punctuation characters.
- `%s` represents all space characters.
- `%u` represents all uppercase letters.
- `%w` represents all alphanumeric characters.
- `%x` represents all hexadecimal digits.
- `%z` represents the character with representation `0`.
- `%x` (where x is any non-alphanumeric character) represents the character x. This is the standard way to escape the magic characters. Any punctuation character (even the non-magic) can be preceded by a `%` when used to represent itself in a pattern.
- `[set]` represents the class which is the union of all characters in `set`. A range of characters may be specified by separating the end characters of the range with a `-`. All classes `%x` described above may also be used as components in `set`. All other characters in `set` represent themselves. For example, `[%w_]` (or `[_%w]`) represents all alphanumeric characters plus the underscore, `[0-7]` represents the octal digits, and `[0-7%l%-]` represents the octal digits plus the lowercase letters plus the `-` character.

The interaction between ranges and classes is not defined. Therefore, patterns like `[%a-z]` or `[a-%%]` have no meaning.

- `[^set]` represents the complement of `set`, where `set` is interpreted as above.

For all classes represented by single letters (`%a`, `%c`, etc.), the corresponding uppercase letter represents the complement of the class. For instance, `%S` represents all non-space characters.

The definitions of letter, space, and other character groups depend on the current locale. In particular, the class `[a-z]` may not be equivalent to `%l`.

PATTERN ITEM lua-patternitem

A pattern item may be

- a single character class, which matches any single character in the class;
- a single character class followed by *, which matches 0 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by +, which matches 1 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence; lua-nongreedy
- a single character class followed by `-`, which also matches 0 or more repetitions of characters in the class. Unlike *, these repetition items will always match the shortest possible sequence;
- a single character class followed by ?, which matches 0 or 1 occurrences of a character in the class;
- `%n`, for n between 1 and 9; such item matches a substring equal to the n -th captured string (see below);
- `%bxy`, where x and y are two distinct characters; such item matches strings that start with x, end with y, and where the x and y are balanced. This means that, if one reads the string from left to right, counting +1 for an x and -1 for a y, the ending y is the first y where the count reaches 0. For instance, the item `%b()` matches expressions with balanced parentheses.

PATTERN

A pattern is a sequence of pattern items. A ^ at the beginning of a pattern anchors the match at the beginning of the subject string. A $ at the end of a pattern anchors the match at the end of the subject string. At other positions, ^ and $ have no special meaning and represent themselves.

CAPTURES lua-capture

A pattern may contain sub-patterns enclosed in parentheses; they describe captures. When a match succeeds, the substrings of the subject string that match captures are stored (captured) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern `"(a*(.)%w(%s*))"`, the part of the string matching `"a*(.)%w(%s*)"` is stored as the first capture (and therefore has number 1); the character matching `.` is captured with number 2, and the part matching `%s*` has number 3.

As a special case, the empty capture `()` captures the current string position (a number). For instance, if we apply the pattern `"()aa()"` on the string `"flaaap"`, there will be two captures: 3 and 5.

A pattern cannot contain embedded zeros. Use `%z` instead.

5.5 Table Manipulation lua-lib-table

This library provides generic functions for table manipulation. It provides all its functions inside the table `table`.

Most functions in the table library assume that the table represents an array or a list. For those functions, when we talk about the "length" of a table we mean the result of the length operator.

table.concat({table} [, {sep} [, {i} [, {j}]]]) table.concat()
Given an array where all elements are strings or numbers, returns `table[i]..sep..table[i+1] ... sep..table[j]`. The default value for `{sep}` is the empty string, the default for `{i}` is 1, and the default for `{j}` is the length of the table. If `{i}` is greater than `{j}`, returns the empty string.

table.foreach({table}, {f}) table.foreach()
Executes the given {f} over all elements of {table}. For each element, {f} is called with the index and respective value as arguments. If {f} returns a non-`nil` value, then the loop is broken, and this value is returned as the final value of table.foreach.

See next() for extra information about table traversals.

table.foreachi({table}, {f}) table.foreachi()
Executes the given {f} over the numerical indices of {table}. For each index, {f} is called with the index and respective value as arguments. Indices are visited in sequential order, from 1 to n, where n is the length of the table. If {f} returns a non-`nil` value, then the loop is broken and this value is returned as the result of table.foreachi.

table.insert({table}, [{pos},] {value}) table.insert()
Inserts element {value} at position {pos} in {table}, shifting up other elements to open space, if necessary. The default value for {pos} is n+1, where n is the length of the table (see lua-length), so that a call table.insert(t,x) inserts x at the end of table t.

table.maxn({table}) table.maxn()
Returns the largest positive numerical index of the given table, or zero if the table has no positive numerical indices. (To do its job this function does a linear traversal of the whole table.)

table.remove({table} [, {pos}]) table.remove()
Removes from {table} the element at position {pos}, shifting down other elements to close the space, if necessary. Returns the value of the removed element. The default value for {pos} is n, where n is the length of the table (see lua-length), so that a call table.remove(t) removes the last element of table t.

table.sort({table} [, {comp}]) table.sort()
Sorts table elements in a given order, in-place, from table[1] to table[n], where n is the length of the table (see lua-length). If {comp} is given, then it must be a function that receives two table elements, and returns true when the first is less than the second (so that not comp(a[i+1],a[i]) will be true after the sort). If {comp} is not given, then the standard Lua operator < is used instead.

The sort algorithm is not stable, that is, elements considered equal by the given order may have their relative positions changed by the sort.

5.6 Mathematical Functions lua-lib-math

This library is an interface to most of the functions of the standard C math library. It provides all its functions inside the table math.

math.abs({x}) math.abs()
Returns the absolute value of {x}.

math.acos({x}) math.acos()
Returns the arc cosine of {x} (in radians).

math.asin({x}) math.asin()
Returns the arc sine of {x} (in radians).

math.atan({x}) math.atan()
Returns the arc tangent of {x} (in radians).

math.atan2({x}, {y}) math.atan2()
Returns the arc tangent of x/y (in radians), but uses the signs of both parameters to find the quadrant of the result. (It also handles correctly the case of {y} being zero.)

math.ceil({x}) math.ceil()
Returns the smallest integer larger than or equal to {x}.

math.cos({x}) math.cos()
Returns the cosine of {x} (assumed to be in radians).

math.cosh({x}) math.cosh()
Returns the hyperbolic cosine of {x}.

math.deg({x}) math.deg()
Returns the angle {x} (given in radians) in degrees.

math.exp({x}) math.exp()
Returns the value e^x.

math.floor({x}) math.floor()
Returns the largest integer smaller than or equal to {x}.

math.fmod({x}, {y}) math.fmod()
Returns the remainder of the division of {x} by {y}.

math.frexp({x}) math.frexp()
Returns m and e such that x = m * 2^e, e is an integer and the absolute value of m is in the range [0.5, 1) (or zero when {x} is zero).

math.huge math.huge
The value HUGE_VAL, a value larger than or equal to any other numerical value.

math.ldexp({m}, {e}) math.ldexp()
Returns m * 2^e (e should be an integer).

math.log({x}) math.log()
Returns the natural logarithm of {x}.

math.log10({x}) math.log10()
Returns the base-10 logarithm of {x}.

math.max({x}, {...}) math.max()
Returns the maximum value among its arguments.

math.min({x}, {...}) math.min()
Returns the minimum value among its arguments.

math.modf({x}) math.modf()
Returns two numbers, the integral part of {x} and the fractional part of {x}.

math.pi math.pi
The value of pi.

math.pow({x}, {y}) math.pow()
Returns x^y. (You can also use the expression x^y to compute this value.)

math.rad({x}) math.rad()
Returns the angle {x} (given in degrees) in radians.

math.random([{m} [, {n}]]) math.random()
This function is an interface to the simple pseudo-random generator function rand provided by ANSI C. (No guarantees can be given for its statistical properties.)

When called without arguments, returns a pseudo-random real number in the range [0,1). When called with a number {m}, math.random returns a pseudo-random integer in the range [1, m]. When called with two numbers {m} and {n}, math.random returns a pseudo-random integer in the range [m, n].

math.randomseed({x}) math.randomseed()
Sets {x} as the "seed" for the pseudo-random generator: equal seeds produce equal sequences of numbers.

math.sin({x}) math.sin()
Returns the sine of {x} (assumed to be in radians).

math.sinh({x}) math.sinh()
Returns the hyperbolic sine of {x}.

math.sqrt({x}) math.sqrt()
Returns the square root of {x}. (You can also use the expression x^0.5 to compute this value.)

math.tan({x}) math.tan()
Returns the tangent of {x} (assumed to be in radians).

math.tanh({x}) math.tanh()
Returns the hyperbolic tangent of {x}.

5.6 Input and Output Facilities lua-lib-io

The I/O library provides two different styles for file manipulation. The first one uses implicit file descriptors; that is, there are operations to set a default input file and a default output file, and all input/output operations are over these default files. The second style uses explicit file descriptors.

When using implicit file descriptors, all operations are supplied by table io. When using explicit file descriptors, the operation io.open returns a file descriptor and then all operations are supplied as methods of the file descriptor.

The table io also provides three predefined file descriptors with their usual meanings from C: io.stdin, io.stdout, and io.stderr.

Unless otherwise stated, all I/O functions return nil on failure (plus an error message as a second result) and some value different from nil on success.

io.close([{file}]) io.close()
Equivalent to file:close. Without a {file}, closes the default output file.

io.flush() io.flush()
Equivalent to file:flush over the default output file.

io.input([{file}]) io.input()
When called with a file name, it opens the named file (in text mode), and sets its handle as the default input file. When called with a file handle, it simply sets this file handle as the default input file. When called without parameters, it returns the current default input file.

In case of errors this function raises the error, instead of returning an error code.

io.lines([{filename}]) io.lines()
Opens the given file name in read mode and returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

```
for line in io.lines(filename) do body end
```

will iterate over all lines of the file. When the iterator function detects the end of file, it returns nil (to finish the loop) and automatically closes the file.

The call io.lines() (without a file name) is equivalent to io.input():lines(); that is, it iterates over the lines of the default input file. In this case it does not close the file when the loop ends.

io.open({filename} [, {mode}]) io.open()
This function opens a file, in the mode specified in the string {mode}. It returns a new file handle, or, in case of errors, nil plus an error message.

The {mode} string can be any of the following:

- "r" read mode (the default);
- "w" write mode;
- "a" append mode;
- "r+" update mode, all previous data is preserved;
- "w+" update mode, all previous data is erased;
- "a+" append update mode, previous data is preserved, writing is only allowed at the end of file.

The {mode} string may also have a b at the end, which is needed in some systems to open the file in binary mode. This string is exactly what is used in the standard C function fopen.

io.output([{file}]) io.output()
Similar to io.input, but operates over the default output file.

io.popen({prog} [, {mode}]) io.popen()
Starts program {prog} in a separated process and returns a file handle that you can use to read data from this program (if {mode} is "r", the default) or to write data to this program (if {mode} is "w").

This function is system dependent and is not available on all platforms.

io.read({...}) io.read()
Equivalent to io.input():read.

io.tmpfile() io.tmpfile()
Returns a handle for a temporary file. This file is opened in update mode and it is automatically removed when the program ends.

io.type({obj}) io.type()
Checks whether {obj} is a valid file handle. Returns the string "file" if {obj} is an open file handle, "closed file" if {obj} is a closed file handle, or nil if {obj} is not a file handle.

io.write({...}) io.write()
Equivalent to io.output():write.

file:close() file:close()
Closes file. Note that files are automatically closed when their handles are garbage collected, but that takes an unpredictable amount of time to happen.

file:flush() file:flush()
Saves any written data to file.

file:lines() file:lines()
Returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

```
for line in file:lines() do body end
```

will iterate over all lines of the file. (Unlike io.lines, this function does not close the file when the loop ends.)

file:read({...}) file:read()
Reads the file file, according to the given formats, which specify what to read. For each format, the function returns a string (or a number) with the characters read, or nil if it cannot read data with the specified format. When called without formats, it uses a default format that reads the entire next line (see below).

The available formats are

"*n" reads a number; this is the only format that returns a number instead of a string. "*a" reads the whole file, starting at the current position. On end of file, it returns the empty string. "*l" reads the next line (skipping the end of line), returning nil on end of file. This is the default format. number reads a string with up to that number of characters, returning nil on end of file. If number is zero, it reads nothing and returns an empty string, or nil on end of file.

file:seek([{whence}] [, {offset}]) file:seek()
Sets and gets the file position, measured from the beginning of the file, to the position given by {offset} plus a base specified by the string {whence}, as follows:

- "set": base is position 0 (beginning of the file);
- "cur": base is current position;
- "end": base is end of file;

In case of success, function seek returns the final file position, measured in bytes from the beginning of the file. If this function fails, it returns nil, plus a string describing the error.

The default value for {whence} is "cur", and for {offset} is 0. Therefore, the call file:seek() returns the current file position, without changing it; the call file:seek("set") sets the position to the beginning of the file (and returns 0); and the call file:seek("end") sets the position to the end of the file, and returns its size.

file:setvbuf({mode} [, {size}]) file:setvbuf()
Sets the buffering mode for an output file. There are three available modes:

"no" no buffering; the result of any output operation appears immediately. "full" full buffering; output operation is performed only when the buffer is full (or when you explicitly flush the file (see io.flush()). "line" line buffering; output is buffered until a newline is output or there is any input from some special files (such as a terminal device).

For the last two cases, {size} specifies the size of the buffer, in bytes. The default is an appropriate size.

file:write({...}) file:write()
Writes the value of each of its arguments to file. The arguments must be strings or numbers. To write other values, use tostring tostring() or string.format string.format() before write.

5.8 Operating System Facilities lua-lib-os

This library is implemented through table os.

os.clock() os.clock()
Returns an approximation of the amount in seconds of CPU time used by the program.

os.date([{format} [, {time}]]) os.date()
Returns a string or a table containing date and time, formatted according to the given string {format}.

If the {time} argument is present, this is the time to be formatted (see the os.time function os.time() for a description of this value). Otherwise, date formats the current time.

If {format} starts with !, then the date is formatted in Coordinated Universal Time. After this optional character, if {format} is the string "*t", then date returns a table with the following fields: year (four digits), month (1-12), day (1-31), hour (0-23), min (0-59), sec (0-61), wday (weekday, Sunday is 1), yday (day of the year), and isdst (daylight saving flag, a boolean).

If {format} is not "*t", then date returns the date as a string, formatted according to the same rules as the C function strftime.

When called without arguments, date returns a reasonable date and time representation that depends on the host system and on the current locale (that is, os.date() is equivalent to os.date("%c")).

os.difftime({t2}, {t1}) os.difftime()
Returns the number of seconds from time {t1} to time {t2}. In POSIX, Windows, and some other systems, this value is exactly t2 - t1 .

os.execute([{command}]) os.execute()
This function is equivalent to the C function system. It passes {command} to be executed by an operating system shell. It returns a status code, which is system-dependent. If {command} is absent, then it returns nonzero if a shell is available and zero otherwise.

os.exit([{code}]) os.exit()
Calls the C function exit, with an optional {code}, to terminate the host program. The default value for {code} is the success code.

os.getenv({varname}) os.getenv()
Returns the value of the process environment variable {varname}, or nil if the variable is not defined.

os.remove({filename}) os.remove()
Deletes the file with the given name. Directories must be empty to be removed. If this function fails, it returns nil, plus a string describing the error.

os.rename({oldname}, {newname}) os.rename()
Renames file named {oldname} to {newname}. If this function fails, it returns nil, plus a string describing the error.

os.setlocale({locale} [, {category}]) os.setlocale()
Sets the current locale of the program. {locale} is a string specifying a locale; {category} is an optional string describing which category to change: "all", "collate", "ctype", "monetary", "numeric", or "time"; the default category is "all". The function returns the name of the new locale, or nil if the request cannot be honored.

os.time([{table}]) os.time()
Returns the current time when called without arguments, or a time representing the date and time specified by the given table. This table must have fields year, month, and day, and may have fields hour, min, sec, and isdst (for a description of these fields, see the os.date function os.date()).

The returned value is a number, whose meaning depends on your system. In POSIX, Windows, and some other systems, this number counts the number of seconds since some given start time (the "epoch"). In other systems, the meaning is not specified, and the number returned by time can be used only as an argument to date and difftime.

os.tmpname() os.tmpname()
Returns a string with a file name that can be used for a temporary file. The file must be explicitly opened before its use and explicitly removed when no longer needed.

5.9 The Debug Library lua-lib-debug

This library provides the functionality of the debug interface to Lua programs. You should exert care when using this library. The functions provided here should be used exclusively for debugging and similar tasks, such as profiling. Please resist the temptation to use them as a usual programming tool: they can be very slow. Moreover, several of its functions violate some assumptions about Lua code (e.g., that variables local to a function cannot be accessed from outside or that userdata metatables cannot be changed by Lua code) and therefore can compromise otherwise secure code.

All functions in this library are provided inside the debug table. All functions that operate over a thread have an optional first argument which is the thread to operate over. The default is always the current thread.

debug.debug() debug.debug()
Enters an interactive mode with the user, running each string that the user enters. Using simple commands and other debug facilities, the user can inspect global and local variables, change their values, evaluate expressions, and so on. A line containing only the word cont finishes this function, so that the caller continues its execution.

Note that commands for debug.debug are not lexically nested within any function, and so have no direct access to local variables.

debug.getfenv(o) debug.getfenv()
Returns the environment of object {o}.

debug.gethook([{thread}]) debug.gethook()
Returns the current hook settings of the thread, as three values: the current hook function, the current hook mask, and the current hook count (as set by the debug.sethook function).

debug.getinfo([{thread},] {function} [, {what}]) debug.getinfo()
Returns a table with information about a function. You can give the function directly, or you can give a number as the value of {function}, which means the function running at level {function} of the call stack of the given thread: level 0 is the current function (getinfo itself); level 1 is the function that called getinfo; and so on. If {function} is a number larger than the number of active functions, then getinfo returns nil.

The returned table may contain all the fields returned by lua_getinfo (see lua_getinfo()), with the string {what} describing which fields to fill in. The default for {what} is to get all information available, except the table of valid lines. If present, the option f adds a field named func with the function itself. If present, the option L adds a field named activelines with the table of valid lines.

For instance, the expression debug.getinfo(1,"n").name returns the name of the current function, if a reasonable name can be found, and debug.getinfo(print) returns a table with all available information about the print function.

debug.getlocal([{thread},] {level}, {local}) debug.getlocal()
This function returns the name and the value of the local variable with index {local} of the function at level {level} of the stack. (The first parameter or local variable has index 1, and so on, until the last active local variable.) The function returns nil if there is no local variable

with the given index, and raises an error when called with a {level} out of range. (You can call debug.getinfo debug.getinfo() to check whether the level is valid.)

Variable names starting with ((open parentheses) represent internal variables (loop control variables, temporaries, and C function locals).

debug.getmetatable({object}) debug.getmetatable()
Returns the metatable of the given {object} or nil if it does not have a metatable.

debug.getregistry() debug.getregistry()
Returns the registry table (see lua-registry).

debug.getupvalue({func}, {up}) debug.getupvalue()
This function returns the name and the value of the upvalue with index {up} of the function {func}. The function returns nil if there is no upvalue with the given index.

debug.setfenv({object}, {table}) debug.setfenv()
Sets the environment of the given {object} to the given {table}. Returns {object}.

debug.sethook([{thread},] {hook}, {mask} [, {count}]) debug.sethook()
Sets the given function as a hook. The string {mask} and the number {count} describe when the hook will be called. The string mask may have the following characters, with the given meaning:

- "c" : The hook is called every time Lua calls a function;
- "r" : The hook is called every time Lua returns from a function;
- "l" : The hook is called every time Lua enters a new line of code.

With a {count} different from zero, the hook is called after every {count} instructions.

When called without arguments, the debug.sethook turns off the hook.

When the hook is called, its first parameter is a string describing the event that triggered its call: "call", "return" (or "tail return"), "line", and "count". For line events, the hook also gets the new line number as its second parameter. Inside a hook, you can call getinfo with level 2 to get more information about the running function (level 0 is the getinfo function, and level 1 is the hook function), unless the event is "tail return". In this case, Lua is only simulating the return, and a call to getinfo will return invalid data.

debug.setlocal([{thread},] {level}, {local}, {value}) debug.setlocal()
This function assigns the value {value} to the local variable with index {local} of the function at level {level} of the stack. The function returns nil if there is no local variable with the given index, and raises an error when called with a {level} out of range. (You can call getinfo to check whether the level is valid.) Otherwise, it returns the name of the local variable.

debug.setmetatable({object}, {table}) debug.setmetatable()
Sets the metatable for the given {object} to the given {table} (which can be nil).

debug.setupvalue({func}, {up}, {value}) debug.setupvalue()
This function assigns the value {value} to the upvalue with index {up} of the function {func}. The function returns nil if there is no upvalue with the given index. Otherwise, it returns the name of the upvalue.

debug.traceback([{thread},] [{message} [,{level}]]) debug.traceback()
Returns a string with a traceback of the call stack. An optional {message} string is appended at the beginning of the traceback. An optional {level} number tells at which level to start the traceback (default is 1, the function calling traceback).

A BIBLIOGRAPHY lua-ref-bibliography

This help file is a minor adaptation from this main reference:

- R. Ierusalimschy, L. H. de Figueiredo, and W. Celes., "Lua: 5.1 reference manual", https://www.lua.org/manual/5.1/manual.html

Lua is discussed in these references:

- R. Ierusalimschy, L. H. de Figueiredo, and W. Celes., "Lua --- an extensible extension language". "Software: Practice & Experience" 26, 6 (1996) 635-652.
- L. H. de Figueiredo, R. Ierusalimschy, and W. Celes., "The design and implementation of a language for extending applications". "Proc. of XXI Brazilian Seminar on Software and Hardware" (1994) 273-283.
- L. H. de Figueiredo, R. Ierusalimschy, and W. Celes., "Lua: an extensible embedded language". "Dr. Dobb's Journal" 21, 12 (Dec 1996) 26-33.
- R. Ierusalimschy, L. H. de Figueiredo, and W. Celes., "The evolution of an extension language: a history of Lua". "Proc. of V Brazilian Symposium on Programming Languages" (2001) B-14-B-28.

B COPYRIGHT AND LICENSES lua-ref-copyright

This help file has the same copyright and license as Lua 5.1 and the Lua 5.1 manual:

Copyright (c) 1994-2006 Lua.org, PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C LUAREF DOC

lua-ref-doc

This is a Vim help file containing a reference for Lua 5.1, and it is -- with a few exceptions and adaptations -- a copy of the Lua 5.1 Reference Manual (see lua-ref-bibliography). For usage information, refer to lua-ref-doc. For copyright information, see lua-ref-copyright.

The main ideas and concepts on how to implement this reference were taken from Christian Habermann's CRefVim project (https://www.vim.org/scripts/script.php?script_id=614).

Adapted for bundled Nvim documentation; the original plugin can be found at https://www.vim.org/scripts/script.php?script_id=1291

LUV REFERENCE MANUAL

This file documents the Lua bindings for the LibUV library which is used for Nvim's event-loop and is accessible from Lua via vim.uv (e.g., uv.version() is exposed as vim.uv.version()).

For information about this manual, see luv-credits.

For further examples, see https://github.com/luvit/luv/tree/master/examples.

INTRODUCTION

luv luv-intro uv

The luv (https://github.com/luvit/luv) project provides access to the multi-platform support library libuv (https://github.com/libuv/libuv) in Lua code. It was primarily developed for the luvit (https://github.com/luvit/luvit) project as the built-in uv module, but can be used in other Lua environments.

More information about the core libuv library can be found at the original libuv documentation page (https://docs.libuv.org/).

TCP Echo Server Example

Here is a small example showing a TCP echo server:

```
local uv = vim.uv

local server = uv.new_tcp()
assert(server:bind("127.0.0.1", 1337))
server:listen(128, function (err)
  assert(not err, err)
  local client = uv.new_tcp()
  server:accept(client)
  client:read_start(function (err, chunk)
    assert(not err, err)
    if chunk then
      client:write(chunk)
    else
      client:shutdown()
      client:close()
    end
  end)
end)
print("TCP server listening at 127.0.0.1 port 1337")
uv.run() -- an explicit run call is necessary outside of luvit
```

Module Layout

The luv library contains a single Lua module referred to hereafter as uv for simplicity. This module consists mostly of functions with names corresponding to their original libuv versions. For example, the libuv function uv_tcp_bind has a luv version at uv.tcp_bind(). Currently, only two non-function fields exists: uv.constants and uv.errno, which are tables.

Functions vs Methods

In addition to having simple functions, luv provides an optional method-style API. For example, uv.tcp_bind(server, host, port) can alternatively be called as server:bind(host, port) . Note that the first argument server becomes the object and tcp_ is removed from the function name. Method forms are documented below where they exist.

Synchronous vs Asynchronous Functions

Functions that accept a callback are asynchronous. These functions may immediately return results to the caller to indicate their initial status, but their final execution is deferred until at least the next libuv loop iteration. After completion, their callbacks are executed with any results passed to it.

Functions that do not accept a callback are synchronous. These functions immediately return their results to the caller.

Some (generally FS and DNS) functions can behave either synchronously or asynchronously. If a callback is provided to these functions, they behave asynchronously; if no callback is provided, they behave synchronously.

Pseudo-Types

Some unique types are defined. These are not actual types in Lua, but they are used here to facilitate documenting consistent behavior:

- fail: an assertable nil, string, string tuple (see luv-error-handling)
- callable: a function; or a table or userdata with a __call metamethod
- buffer: a string or a sequential table of strings
- threadargs: variable arguments (...) of type nil, boolean, number, string, or userdata; number of arguments limited to 9.

CONTENTS

luv-contents

This documentation is mostly a retelling of the libuv API documentation (https://docs.libuv.org/en/v1.x/api.html) within the context of luv's Lua API. Low-level implementation details and unexposed C functions and types are not documented here except for when they are relevant to behavior seen in the Lua module.

- luv-constants — Constants
- luv-error-handling — Error handling
- luv-version-checking — Version checking
- uv_loop_t — Event loop
- uv_req_t — Base request
- uv_handle_t — Base handle
  - uv_timer_t — Timer handle
  - uv_prepare_t — Prepare handle
  - uv_check_t — Check handle
  - uv_idle_t — Idle handle
  - uv_async_t — Async handle
  - uv_poll_t — Poll handle
  - uv_signal_t — Signal handle
  - uv_process_t — Process handle
  - uv_stream_t — Stream handle
    - uv_tcp_t — TCP handle
    - uv_pipe_t — Pipe handle
    - uv_tty_t — TTY handle
  - uv_udp_t — UDP handle
  - uv_fs_event_t — FS Event handle
  - uv_fs_poll_t — FS Poll handle
- luv-file-system-operations — File system operations
- luv-thread-pool-work-scheduling — Thread pool work scheduling
- luv-dns-utility-functions — DNS utility functions
- luv-threading-and-synchronization-utilities — Threading and synchronization utilities
- luv-miscellaneous-utilities — Miscellaneous utilities
- luv-metrics-operations — Metrics operations

CONSTANTS

luv-constants

As a Lua library, luv supports and encourages the use of lowercase strings to represent options. For example:

```
-- signal start with string input
uv.signal_start("sigterm", function(signame)
  print(signame) -- string output: "sigterm"
end)
```

However, luv also superficially exposes libuv constants in a Lua table at uv.constants where its keys are uppercase constant names and their associated values are integers defined internally by libuv. The values from this table may be supported as function arguments, but their use may not change the output type. For example:

```
-- signal start with integer input
uv.signal_start(uv.constants.SIGTERM, function(signame)
  print(signame) -- string output: "sigterm"
end)
```

The uppercase constants defined in uv.constants that have associated lowercase option strings are listed below.

Address Families

- AF_UNIX: "unix"
- AF_INET: "inet"
- AF_INET6: "inet6"
- AF_IPX: "ipx"
- AF_NETLINK: "netlink"
- AF_X25: "x25"
- AF_AX25: "as25"
- AF_ATMPVC: "atmpvc"
- AF_APPLETALK: "appletalk"
- AF_PACKET: "packet"

Signals

- SIGHUP: "sighup"
- SIGINT: "sigint"
- SIGQUIT: "sigquit"
- SIGILL: "sigill"
- SIGTRAP: "sigtrap"
- SIGABRT: "sigabrt"
- SIGIOT: "sigiot"
- SIGBUS: "sigbus"
- SIGFPE: "sigfpe"
- SIGKILL: "sigkill"
- SIGUSR1: "sigusr1"

- SIGSEGV: "sigsegv"
- SIGUSR2: "sigusr2"
- SIGPIPE: "sigpipe"
- SIGALRM: "sigalrm"
- SIGTERM: "sigterm"
- SIGCHLD: "sigchld"
- SIGSTKFLT: "sigstkflt"
- SIGCONT: "sigcont"
- SIGSTOP: "sigstop"
- SIGTSTP: "sigtstp"
- SIGBREAK: "sigbreak"
- SIGTTIN: "sigttin"
- SIGTTOU: "sigttou"
- SIGURG: "sigurg"
- SIGXCPU: "sigxcpu"
- SIGXFSZ: "sigxfsz"
- SIGVTALRM: "sigvtalrm"
- SIGPROF: "sigprof"
- SIGWINCH: "sigwinch"
- SIGIO: "sigio"
- SIGPOLL: "sigpoll"
- SIGLOST: "siglost"
- SIGPWR: "sigpwr"
- SIGSYS: "sigsys"

Socket Types

- SOCK_STREAM: "stream"
- SOCK_DGRAM: "dgram"
- SOCK_SEQPACKET: "seqpacket"
- SOCK_RAW: "raw"
- SOCK_RDM: "rdm"

TTY Modes

- TTY_MODE_NORMAL: "normal"
- TTY_MODE_RAW: "raw"
- TTY_MODE_IO: "io"

TTY_MODE_RAW_VT: "raw_vt"

FS Modification Times

- FS_UTIME_NOW: "now"
- FS_UTIME_OMIT: "omit"

ERROR HANDLING

luv-error-handling

In libuv, errors are represented by negative numbered constants. While these constants are made available in the uv.errno table, they are not returned by luv functions and the libuv functions used to handle them are not exposed. Instead, if an internal error is encountered, the failing luv function will return to the caller an assertable nil, err, name tuple:

- nil idiomatically indicates failure
- err is a string with the format {name}: {message}
  - {name} is the error name provided internally by uv_err_name
  - {message} is a human-readable message provided internally by uv_strerror
- name is the same string used to construct err

This tuple is referred to below as the fail pseudo-type.

When a function is called successfully, it will return either a value that is relevant to the operation of the function, or the integer 0 to indicate success, or sometimes nothing at all. These cases are documented below.

uv.errno

uv.errno

Below is a list of known error names and error strings. See Libuv's "Error constants" page for further details. (https://docs.libuv.org/en/v1.x/errors.html#error-constants)

- E2BIG: argument list too long.
- EACCES: permission denied.
- EADDRINUSE: address already in use.
- EADDRNOTAVAIL: address not available.
- EAFNOSUPPORT: address family not supported.
- EAGAIN: resource temporarily unavailable.
- EAI_ADDRFAMILY: address family not supported.
- EAI_AGAIN: temporary failure.
- EAI_BADFLAGS: bad ai_flags value.
- EAI_BADHINTS: invalid value for hints.
- EAI_CANCELED: request canceled.
- EAI_FAIL: permanent failure.

- EAI_FAMILY: ai_family not supported.
- EAI_MEMORY: out of memory.
- EAI_NODATA: no address.
- EAI_NONAME: unknown node or service.
- EAI_OVERFLOW: argument buffer overflow.
- EAI_PROTOCOL: resolved protocol is unknown.
- EAI_SERVICE: service not available for socket type.
- EAI_SOCKTYPE: socket type not supported.
- EALREADY: connection already in progress.
- EBADF: bad file descriptor.
- EBUSY: resource busy or locked.
- ECANCELED: operation canceled.
- ECHARSET: invalid Unicode character.
- ECONNABORTED: software caused connection abort.
- ECONNREFUSED: connection refused.
- ECONNRESET: connection reset by peer.
- EDESTADDRREQ: destination address required.
- EEXIST: file already exists.
- EFAULT: bad address in system call argument.
- EFBIG: file too large.
- EHOSTUNREACH: host is unreachable.
- EINTR: interrupted system call.
- EINVAL: invalid argument.
- EIO: i/o error.
- EISCONN: socket is already connected.
- EISDIR: illegal operation on a directory.
- ELOOP: too many symbolic links encountered.
- EMFILE: too many open files.
- EMSGSIZE: message too long.
- ENAMETOOLONG: name too long.

- ENETDOWN: network is down.
- ENETUNREACH: network is unreachable.
- ENFILE: file table overflow.
- ENOBUFS: no buffer space available.
- ENODEV: no such device.
- ENOENT: no such file or directory.
- ENOMEM: not enough memory.
- ENONET: machine is not on the network.
- ENOPROTOOPT: protocol not available.
- ENOSPC: no space left on device.
- ENOSYS: function not implemented.
- ENOTCONN: socket is not connected.
- ENOTDIR: not a directory.
- ENOTEMPTY: directory not empty.
- ENOTSOCK: socket operation on non-socket.
- ENOTSUP: operation not supported on socket.
- EOVERFLOW: value too large for defined data type.
- EPERM: operation not permitted.
- EPIPE: broken pipe.
- EPROTO: protocol error.
- EPROTONOSUPPORT: protocol not supported.
- EPROTOTYPE: protocol wrong type for socket.
- ERANGE: result too large.
- EROFS: read-only file system.
- ESHUTDOWN: cannot send after transport endpoint shutdown.
- ESPIPE: invalid seek.
- ESRCH: no such process.
- ETIMEDOUT: connection timed out.
- ETXTBSY: text file is busy.
- EXDEV: cross-device link not permitted.

- UNKNOWN: unknown error.
- EOF: end of file.
- ENXIO: no such device or address.
- EMLINK: too many links.
- ENOTTY: inappropriate ioctl for device.
- EFTYPE: inappropriate file type or format.
- EILSEQ: illegal byte sequence.
- ESOCKTNOSUPPORT: socket type not supported.

VERSION CHECKING

luv-version-checking

uv.version()

uv.version()

Returns the libuv version packed into a single integer. 8 bits are used for each component, with the patch number stored in the 8 least significant bits. For example, this would be 0x010203 in libuv 1.2.3.

Returns: integer

uv.version_string()

uv.version_string()

Returns the libuv version number as a string. For example, this would be "1.2.3" in libuv 1.2.3. For non-release versions, the version suffix is included.

Returns: string

uv_loop_t — Event loop

luv-event-loop uv_loop_t

The event loop is the central part of libuv's functionality. It takes care of polling for I/O and scheduling callbacks to be run based on different sources of events.

In luv, there is an implicit uv loop for every Lua state that loads the library. You can use this library in an multi-threaded environment as long as each thread has it's own Lua state with its corresponding own uv loop. This loop is not directly exposed to users in the Lua module.

uv.loop_close()

uv.loop_close()

Closes all internal loop resources. In normal execution, the loop will automatically be closed when it is garbage collected by Lua, so it is not necessary to explicitly call loop_close(). Call this function only after the loop has finished executing and all open handles and requests have been closed, or it will return EBUSY.

Returns: 0 or fail

uv.run([{mode}])

uv.run()

Parameters:

- mode: string or nil (default: "default")

This function runs the event loop. It will act differently depending on the specified mode:

- "default": Runs the event loop until there are no more active and referenced handles or requests. Returns true if uv.stop() was called and there are still active handles or requests. Returns false in all other cases.
- "once": Poll for I/O once. Note that this function blocks if there are no pending callbacks. Returns false when done (no active handles or requests left), or true if more callbacks are expected (meaning you should run the event loop again sometime in the future).
- "nowait": Poll for I/O once but don't block if there are no pending callbacks. Returns false if done (no active handles or requests left), or true if more callbacks are expected (meaning you should run the event loop again sometime in the future).

Returns: boolean or fail

Note: Luvit will implicitly call uv.run() after loading user code, but if you use the luv bindings directly, you need to call this after registering your initial set of event callbacks to start the event loop.

uv.loop_configure({option}, {...}) uv.loop_configure()

Parameters:

- option: string
- ...: depends on option, see below

Set additional loop options. You should normally call this before the first call to uv_run() unless mentioned otherwise.

Supported options:

- "block_signal": Block a signal when polling for new events. The second argument to loop_configure() is the signal name (as a lowercase string) or the signal number. This operation is currently only implemented for "sigprof" signals, to suppress unnecessary wakeups when using a sampling profiler. Requesting other signals will fail with EINVAL.
- "metrics_idle_time": Accumulate the amount of idle time the event loop spends in the event provider. This option is necessary to use metrics_idle_time().

An example of a valid call to this function is:

```
uv.loop_configure("block_signal", "sigprof")
```

Returns: 0 or fail

Note: Be prepared to handle the ENOSYS error; it means the loop option is not supported by the platform.

uv.loop_mode() uv.loop_mode()

If the loop is running, returns a string indicating the mode in use. If the loop is not running, nil is returned instead.

Returns: string or nil

uv.loop_alive() uv.loop_alive()

Returns true if there are referenced active handles, active requests, or closing handles in the loop; otherwise, false.

Returns: boolean or fail

uv.stop() uv.stop()

Stop the event loop, causing uv.run() to end as soon as possible. This will happen not sooner than the next loop iteration. If this function was called before blocking for I/O, the loop won't block for I/O on this iteration.

Returns: Nothing.

uv.backend_fd() uv.backend_fd()

Get backend file descriptor. Only kqueue, epoll, and event ports are supported.

This can be used in conjunction with uv.run("nowait") to poll in one thread and run the event loop's callbacks in another

Returns: integer or nil

Note: Embedding a kqueue fd in another kqueue pollset doesn't work on all platforms. It's not an error to add the fd but it never generates events.

uv.backend_timeout() uv.backend_timeout()

Get the poll timeout. The return value is in milliseconds, or -1 for no timeout.

Returns: integer

uv.now() uv.now()

Returns the current timestamp in milliseconds. The timestamp is cached at the start of the event loop tick, see uv.update_time() for details and rationale.

The timestamp increases monotonically from some arbitrary point in time. Don't make assumptions about the starting point, you will only get disappointed.

Returns: integer

Note: Use uv.hrtime() if you need sub-millisecond granularity.

uv.update_time() uv.update_time()

Update the event loop's concept of "now". Libuv caches the current time at the start of the event loop tick in order to reduce the number of time-related system calls.

You won't normally need to call this function unless you have callbacks that block the event loop for longer periods of time, where "longer" is somewhat subjective but probably on the order of a millisecond or more.

Returns: Nothing.

uv.walk({callback}) uv.walk()

Parameters:

- callback: callable
    - handle: userdata for sub-type of uv_handle_t

Walk the list of handles: callback will be executed with each handle.

Returns: Nothing.

```
-- Example usage of uv.walk to close all handles that
-- aren't already closing.
uv.walk(function (handle)
  if not handle:is_closing() then
    handle:close()
  end
end)
```

uv_req_t — Base request

luv-base-request uv_req_t

uv_req_t is the base type for all libuv request types.

uv.cancel({req}) uv.cancel()

> method form req:cancel()

Parameters:

- req: userdata for sub-type of uv_req_t

Cancel a pending request. Fails if the request is executing or has finished executing. Only cancellation of uv_fs_t, uv_getaddrinfo_t, uv_getnameinfo_t and uv_work_t requests is currently supported.

Returns: 0 or fail

uv.req_get_type({req}) uv.req_get_type()

> method form req:get_type()

Parameters:

- req: userdata for sub-type of uv_req_t

Returns the name of the struct for a given request (e.g. "fs" for uv_fs_t) and the libuv enum integer for the request's type (uv_req_type).

Returns: string, integer

uv_handle_t — Base handle

luv-base-handle uv_handle_t

uv_handle_t is the base type for all libuv handle types. All API functions defined here work with any handle type.

uv.is_active({handle}) uv.is_active()

> method form handle:is_active()

Parameters:

- handle: userdata for sub-type of uv_handle_t

Returns true if the handle is active, false if it's inactive. What "active" means depends on the type of handle:

- A uv_async_t handle is always active and cannot be deactivated, except by closing it with uv.close().
- A uv_pipe_t, uv_tcp_t, uv_udp_t, etc. handle - basically any handle that deals with I/O - is active when it is doing something that involves I/O, like reading, writing, connecting, accepting new connections, etc.
- A uv_check_t, uv_idle_t, uv_timer_t, etc. handle is active when it has been started with a call

to uv.check_start(), uv.idle_start(), uv.timer_start() etc. until it has been stopped with a call to its respective stop function.

Returns: boolean or fail

uv.is_closing({handle}) uv.is_closing()

> method form handle:is_closing()

Parameters:

- handle: userdata for sub-type of uv_handle_t

Returns true if the handle is closing or closed, false otherwise.

Returns: boolean or fail

Note: This function should only be used between the initialization of the handle and the arrival of the close callback.

uv.close({handle} [, {callback}]) uv.close()

> method form handle:close([callback])

Parameters:

- handle: userdata for sub-type of uv_handle_t
- callback: callable or nil

Request handle to be closed. callback will be called asynchronously after this call. This MUST be called on each handle before memory is released.

Handles that wrap file descriptors are closed immediately but callback will still be deferred to the next iteration of the event loop. It gives you a chance to free up any resources associated with the handle.

In-progress requests, like uv_connect_t or uv_write_t, are cancelled and have their callbacks called asynchronously with ECANCELED.

Returns: Nothing.

uv.ref({handle}) uv.ref()

> method form handle:ref()

Parameters:

- handle: userdata for sub-type of uv_handle_t

Reference the given handle. References are idempotent, that is, if a handle is already referenced calling this function again will have no effect.

Returns: Nothing.

See luv-reference-counting.

uv.unref({handle}) uv.unref()

> method form handle:unref()

Parameters:

- handle: userdata for sub-type of uv_handle_t

Un-reference the given handle. References are idempotent, that is, if a handle is not referenced calling this function again will have no effect.

Returns: Nothing.

See luv-reference-counting.

uv.has_ref({handle}) uv.has_ref()

> method form handle:has_ref()

Parameters:

- handle: userdata for sub-type of uv_handle_t

Returns true if the handle referenced, false if not.

Returns: boolean or fail

See luv-reference-counting.

uv.send_buffer_size({handle} [, {size}]) uv.send_buffer_size()

> method form handle:send_buffer_size([size])

Parameters:

- handle: userdata for sub-type of uv_handle_t
- size: integer or nil (default: 0)

Gets or sets the size of the send buffer that the operating system uses for the socket.

If size is omitted (or 0), this will return the current send buffer size; otherwise, this will use size to set the new send buffer size.

This function works for TCP, pipe and UDP handles on Unix and for TCP and UDP handles on Windows.

Returns:

- integer or fail (if size is nil or 0)
- 0 or fail (if size is not nil and not 0)

Note: Linux will set double the size and return double the size of the original set value.

uv.recv_buffer_size({handle} [, {size}]) uv.recv_buffer_size()

> method form handle:recv_buffer_size([size])

Parameters:

- handle: userdata for sub-type of uv_handle_t
- size: integer or nil (default: 0)

Gets or sets the size of the receive buffer that the operating system uses for the socket.

If size is omitted (or 0), this will return the current send buffer size; otherwise, this will use size to set the new send buffer size.

This function works for TCP, pipe and UDP handles on Unix and for TCP and UDP handles on Windows.

Returns:

- integer or fail (if size is nil or 0)
- 0 or fail (if size is not nil and not 0)

Note: Linux will set double the size and return double the size of the original set value.

uv.fileno({handle}) uv.fileno()

> method form handle:fileno()

Parameters:

- handle: userdata for sub-type of uv_handle_t

Gets the platform dependent file descriptor equivalent.

The following handles are supported: TCP, pipes, TTY, UDP and poll. Passing any other handle type will fail with EINVAL.

If a handle doesn't have an attached file descriptor yet or the handle itself has been closed, this function will return EBADF.

Returns: integer or fail

WARNING: Be very careful when using this function. libuv assumes it's in control of the file descriptor so any change to it may lead to malfunction.

uv.handle_get_type({handle}) uv.handle_get_type()

> method form handle:get_type()

Parameters:

- handle: userdata for sub-type of uv_handle_t

Returns the name of the struct for a given handle (e.g. "pipe" for uv_pipe_t) and the libuv enum integer for the handle's type (uv_handle_type).

Returns: string, integer

REFERENCE COUNTING luv-reference-counting

The libuv event loop (if run in the default mode) will run until there are no active and referenced handles left. The user can force the loop to exit early by unreferencing handles which are active, for example by calling uv.unref() after calling uv.timer_start().

A handle can be referenced or unreferenced, the refcounting scheme doesn't use a counter, so both operations are idempotent.

All handles are referenced when active by default, see uv.is_active() for a more detailed explanation on what being active involves.

uv_timer_t — Timer handle

luv-timer-handle uv_timer_t

> uv_handle_t functions also apply.

Timer handles are used to schedule callbacks to be called in the future.

uv.new_timer() uv.new_timer()

Creates and initializes a new uv_timer_t. Returns the Lua userdata wrapping it.

Returns: uv_timer_t userdata or fail

```
-- Creating a simple setTimeout wrapper
local function setTimeout(timeout, callback)
  local timer = uv.new_timer()
  timer:start(timeout, 0, function ()
    timer:stop()
    timer:close()
    callback()
  end)
  return timer
end

-- Creating a simple setInterval wrapper
local function setInterval(interval, callback)
  local timer = uv.new_timer()
  timer:start(interval, interval, function ()
    callback()
  end)
  return timer
end

-- And clearInterval
local function clearInterval(timer)
  timer:stop()
  timer:close()
end
```

uv.timer_start({timer}, {timeout}, {repeat}, {callback}) uv.timer_start()

> method form timer:start(timeout, repeat, callback)

Parameters:

- timer: uv_timer_t userdata
- timeout: integer
- repeat: integer
- callback: callable

Start the timer. timeout and repeat are in milliseconds.

If timeout is zero, the callback fires on the next event loop iteration. If repeat is non-zero, the callback fires first after timeout milliseconds and then repeatedly after repeat milliseconds.

Returns: 0 or fail

uv.timer_stop({timer}) uv.timer_stop()

> method form timer:stop()

Parameters:

- timer: uv_timer_t userdata

Stop the timer, the callback will not be called anymore.

Returns: 0 or fail

uv.timer_again({timer}) uv.timer_again()

> method form timer:again()

Parameters:

- timer: uv_timer_t userdata

Stop the timer, and if it is repeating restart it using the repeat value as the timeout. If the timer has never been started before it raises EINVAL.

Returns: 0 or fail

uv.timer_set_repeat({timer}, {repeat}) uv.timer_set_repeat()

> method form timer:set_repeat(repeat)

Parameters:

- timer: uv_timer_t userdata

- repeat: integer

Set the repeat interval value in milliseconds. The timer will be scheduled to run on the given interval, regardless of the callback execution duration, and will follow normal timer semantics in the case of a time-slice overrun.

For example, if a 50 ms repeating timer first runs for 17 ms, it will be scheduled to run again 33 ms later. If other tasks consume more than the 33 ms following the first timer callback, then the callback will run as soon as possible.

Returns: Nothing.

uv.timer_get_repeat({timer}) uv.timer_get_repeat()

> method form timer:get_repeat()

Parameters:
- timer: uv_timer_t userdata

Get the timer repeat value.

Returns: integer

uv.timer_get_due_in({timer}) uv.timer_get_due_in()

> method form timer:get_due_in()

Parameters:
- timer: uv_timer_t userdata

Get the timer due value or 0 if it has expired. The time is relative to uv.now().

Returns: integer

Note: New in libuv version 1.40.0.

uv_prepare_t — Prepare handle luv-prepare-handle uv_prepare_t

> uv_handle_t functions also apply.

Prepare handles will run the given callback once per loop iteration, right before polling for I/O.

```
local prepare = uv.new_prepare()
prepare:start(function()
  print("Before I/O polling")
end)
```

uv.new_prepare() uv.new_prepare()

Creates and initializes a new uv_prepare_t. Returns the Lua userdata wrapping it.

Returns: uv_prepare_t userdata

uv.prepare_start({prepare}, {callback}) uv.prepare_start()

> method form prepare:start(callback)

Parameters:
- prepare: uv_prepare_t userdata

- callback: callable

Start the handle with the given callback.

Returns: 0 or fail

uv.prepare_stop({prepare}) uv.prepare_stop()

> method form prepare:stop()

Parameters:
- prepare: uv_prepare_t userdata

Stop the handle, the callback will no longer be called.

Returns: 0 or fail

uv_check_t — Check handle luv-check-handle uv_check_t

> uv_handle_t functions also apply.

Check handles will run the given callback once per loop iteration, right after polling for I/O.

```
local check = uv.new_check()
check:start(function()
  print("After I/O polling")
end)
```

uv.new_check() uv.new_check()

Creates and initializes a new uv_check_t. Returns the Lua userdata wrapping it.

Returns: uv_check_t userdata

uv.check_start({check}, {callback}) uv.check_start()

> method form check:start(callback)

Parameters:

- check: uv_check_t userdata
- callback: callable

Start the handle with the given callback.

Returns: 0 or fail

uv.check_stop({check}) uv.check_stop()

> method form check:stop()

Parameters:

- check: uv_check_t userdata

Stop the handle, the callback will no longer be called.

Returns: 0 or fail

uv_idle_t — Idle handle luv-idle-handle uv_idle_t

> uv_handle_t functions also apply.

Idle handles will run the given callback once per loop iteration, right before the uv_prepare_t handles.

Note: The notable difference with prepare handles is that when there are active idle handles, the loop will perform a zero timeout poll instead of blocking for I/O.

WARNING: Despite the name, idle handles will get their callbacks called on every loop iteration, not when the loop is actually "idle".

```
local idle = uv.new_idle()
idle:start(function()
  print("Before I/O polling, no blocking")
end)
```

uv.new_idle() uv.new_idle()

Creates and initializes a new uv_idle_t. Returns the Lua userdata wrapping it.

Returns: uv_idle_t userdata

uv.idle_start({idle}, {callback}) uv.idle_start()

> method form idle:start(callback)

Parameters:

- idle: uv_idle_t userdata
- callback: callable

Start the handle with the given callback.

Returns: 0 or fail

uv.idle_stop({check}) uv.idle_stop()

> method form idle:stop()

Parameters:

- idle: uv_idle_t userdata

Stop the handle, the callback will no longer be called.

Returns: 0 or fail

uv_async_t — Async handle luv-async-handle uv_async_t

> uv_handle_t functions also apply.

Async handles allow the user to "wakeup" the event loop and get a callback called from another thread.

```
local async
async = uv.new_async(function()
  print("async operation ran")
  async:close()
end)

async:send()
```

uv.new_async({callback}) uv.new_async()

Parameters:

- callback: callable
- ...: threadargs passed to/from uv.async_send(async, ...)

Creates and initializes a new uv_async_t. Returns the Lua userdata wrapping it.

Returns: uv_async_t userdata or fail

Note: Unlike other handle initialization functions, this immediately starts the handle.

uv.async_send({async}, {...}) uv.async_send()

> method form async:send(...)

Parameters:

- async: uv_async_t userdata
- ...: threadargs

Wakeup the event loop and call the async handle's callback.

Returns: 0 or fail

Note: It's safe to call this function from any thread. The callback will be called on the loop thread.

WARNING: libuv will coalesce calls to uv.async_send(async), that is, not every call to it will yield an execution of the callback. For example: if uv.async_send() is called 5 times in a row before the callback is called, the callback will only be called once. If uv.async_send() is called again after the callback was called, it will be called again.

uv_poll_t — Poll handle luv-poll-handle uv_poll_t

> uv_handle_t functions also apply.

Poll handles are used to watch file descriptors for readability and writability, similar to the purpose of poll(2) (https://linux.die.net/man/2/poll).

The purpose of poll handles is to enable integrating external libraries that rely on the event loop to signal it about the socket status changes, like c-ares or libssh2. Using uv_poll_t for any other purpose is not recommended; uv_tcp_t, uv_udp_t, etc. provide an implementation that is faster and more scalable than what can be achieved with uv_poll_t, especially on Windows.

It is possible that poll handles occasionally signal that a file descriptor is readable or writable even when it isn't. The user should therefore always be prepared to handle EAGAIN or equivalent when it attempts to read from or write to the fd.

It is not okay to have multiple active poll handles for the same socket, this can cause libuv to busyloop or otherwise malfunction.

The user should not close a file descriptor while it is being polled by an active poll handle. This can cause the handle to report an error, but it might also start polling another socket. However the fd can be safely closed immediately after a call to uv.poll_stop() or uv.close().

Note: On windows only sockets can be polled with poll handles. On Unix any file descriptor that would be accepted by poll(2) can be used.

uv.new_poll({fd}) uv.new_poll()

Parameters:

- fd: integer

Initialize the handle using a file descriptor.

The file descriptor is set to non-blocking mode.

Returns: uv_poll_t userdata or fail

uv.new_socket_poll({fd}) uv.new_socket_poll()

Parameters:

- fd: integer

Initialize the handle using a socket descriptor. On Unix this is identical to uv.new_poll(). On windows it takes a SOCKET handle.

The socket is set to non-blocking mode.

Returns: uv_poll_t userdata or fail

uv.poll_start({poll}, {events}, {callback}) uv.poll_start()

> method form poll:start(events, callback)

Parameters:

- poll: uv_poll_t userdata
- events: string or nil (default: "rw")
- callback: callable
  - err: nil or string
  - events: string or nil

Starts polling the file descriptor. events are: "r", "w", "rw", "d", "rd", "wd", "rwd", "p", "rp", "wp", "rwp", "dp", "rdp", "wdp", or "rwdp" where r is READABLE, w is WRITABLE, d is DISCONNECT, and p is PRIORITIZED. As soon as an event is detected the callback will be called with status set to 0, and the detected events set on the events field.

The user should not close the socket while the handle is active. If the user does that anyway, the callback may be called reporting an error status, but this is not guaranteed.

Returns: 0 or fail

Note Calling uv.poll_start() on a handle that is already active is fine. Doing so will update the events mask that is being watched for.

uv.poll_stop({poll}) uv.poll_stop()

> method form poll:stop()

Parameters:

- poll: uv_poll_t userdata

Stop polling the file descriptor, the callback will no longer be called.

Returns: 0 or fail

uv_signal_t — Signal handle luv-signal-handle uv_signal_t

> uv_handle_t functions also apply.

Signal handles implement Unix style signal handling on a per-event loop bases.

Windows **Notes:**

Reception of some signals is emulated on Windows:

- SIGINT is normally delivered when the user presses CTRL+C. However, like on Unix, it is not generated when terminal raw mode is enabled.
- SIGBREAK is delivered when the user pressed CTRL + BREAK.
- SIGHUP is generated when the user closes the console window. On SIGHUP the program is given approximately 10 seconds to perform cleanup. After that Windows will unconditionally terminate it.
- SIGWINCH is raised whenever libuv detects that the console has been resized. SIGWINCH is emulated by libuv when the program uses a uv_tty_t handle to write to the console. SIGWINCH may not always be delivered in a timely manner; libuv will only detect size changes when the cursor is being moved. When a readable uv_tty_t handle is used in raw mode, resizing the console buffer will also trigger a SIGWINCH signal.
- Watchers for other signals can be successfully created, but these signals are never received. These signals are: SIGILL, SIGABRT, SIGFPE, SIGSEGV, SIGTERM and SIGKILL.
- Calls to raise() or abort() to programmatically raise a signal are not detected by libuv; these will not trigger a signal watcher.

Unix **Notes:**

- SIGKILL and SIGSTOP are impossible to catch.
- Handling SIGBUS, SIGFPE, SIGILL or SIGSEGV via libuv results into undefined behavior.
- SIGABRT will not be caught by libuv if generated by abort(), e.g. through assert().
- On Linux SIGRT0 and SIGRT1 (signals 32 and 33) are used by the NPTL pthreads library to manage threads. Installing watchers for those signals will lead to unpredictable behavior and is strongly discouraged. Future versions of libuv may simply reject them.

```
-- Create a new signal handler
local signal = uv.new_signal()
-- Define a handler function
uv.signal_start(signal, "sigint", function(signame)
  print("got " .. signame .. ", shutting down")
  os.exit(1)
end)
```

uv.new_signal() uv.new_signal()

Creates and initializes a new uv_signal_t. Returns the Lua userdata wrapping it.

Returns: uv_signal_t userdata or fail

uv.signal_start({signal}, {signame}, {callback}) uv.signal_start()

> method form signal:start(signame, callback)

Parameters:

- signal: uv_signal_t userdata
- signame: string or integer
- callback: callable
    - signame: string

Start the handle with the given callback, watching for the given signal.

See luv-constants for supported signame input and output values.

Returns: 0 or fail

uv.signal_start_oneshot() uv.signal_start_oneshot()

uv.signal_start_oneshot({signal}, {signame}, {callback})

> method form signal:start_oneshot(signame, callback)

Parameters:

- signal: uv_signal_t userdata
- signame: string or integer
- callback: callable
    - signame: string

Same functionality as uv.signal_start() but the signal handler is reset the moment the signal is received.

See luv-constants for supported signame input and output values.

Returns: 0 or fail

uv.signal_stop({signal}) uv.signal_stop()

> method form signal:stop()

Parameters:

- signal: uv_signal_t userdata

Stop the handle, the callback will no longer be called.

Returns: 0 or fail

uv_process_t — Process handle

luv-process-handle uv_process_t

> uv_handle_t functions also apply.

Process handles will spawn a new process and allow the user to control it and establish communication channels with it using streams.

uv.disable_stdio_inheritance()

uv.disable_stdio_inheritance()

Disables inheritance for file descriptors / handles that this process inherited from its parent. The effect is that child processes spawned by this process don't accidentally inherit these handles.

It is recommended to call this function as early in your program as possible, before the inherited file descriptors can be closed or duplicated.

Returns: Nothing.

Note: This function works on a best-effort basis: there is no guarantee that libuv can discover all file descriptors that were inherited. In general it does a better job on Windows than it does on Unix.

uv.spawn({path}, {options}, {on_exit})

uv.spawn()

Parameters:

- path: string
- options: table (see below)
- on_exit: callable
  - code: integer
  - signal: integer

Initializes the process handle and starts the process. If the process is successfully spawned, this function will return the handle and pid of the child process.

Possible reasons for failing to spawn would include (but not be limited to) the file to execute not existing, not having permissions to use the setuid or setgid specified, or not having enough memory to allocate for the new process.

```
local stdin = uv.new_pipe()
local stdout = uv.new_pipe()
local stderr = uv.new_pipe()

print("stdin", stdin)
print("stdout", stdout)
print("stderr", stderr)

local handle, pid = uv.spawn("cat", {
  stdio = {stdin, stdout, stderr}
}, function(code, signal) -- on exit
  print("exit code", code)
  print("exit signal", signal)
end)

print("process opened", handle, pid)

uv.read_start(stdout, function(err, data)
  assert(not err, err)
  if data then
    print("stdout chunk", stdout, data)
  else
    print("stdout end", stdout)
  end
end)

uv.read_start(stderr, function(err, data)
  assert(not err, err)
  if data then
    print("stderr chunk", stderr, data)
  else
    print("stderr end", stderr)
  end
end)

uv.write(stdin, "Hello World")

uv.shutdown(stdin, function()
  print("stdin shutdown", stdin)
  uv.close(handle, function()
    print("process closed", handle, pid)
  end)
end)
```

uv.spawn-options

The options table accepts the following fields:

- options.args - Command line arguments as a list of strings. The first string should not be the path to the program, since that is already provided via path. On Windows, this uses CreateProcess which concatenates the arguments into a string. This can cause some strange errors (see options.verbatim below for Windows).
- options.stdio - Set the file descriptors that will be made available to the child process. The convention is that the first entries are stdin, stdout, and stderr. (**Note:** On Windows, file descriptors after the third are available to the child process only if the child processes uses the MSVCRT runtime.)
- options.env - Set environment variables for the new process. Each entry should be a string in the form of NAME=VALUE.
- options.cwd - Set the current working directory for the sub-process.
- options.uid - Set the child process' user id.
- options.gid - Set the child process' group id.
- options.verbatim - If true, do not wrap any arguments in quotes, or perform any other escaping, when converting the argument list into a command line string. This option is only meaningful on Windows systems. On Unix it is silently ignored.
- options.detached - If true, spawn the child process in a detached state - this will make it a process group leader,

and will effectively enable the child to keep running
after the parent exits. Note that the child process will
still keep the parent's event loop alive unless the parent
process calls uv.unref() on the child's process handle.

- options.hide - If true, hide the subprocess console

window that would normally be created. This option is only
meaningful on Windows systems. On Unix it is silently
ignored.

The options.stdio entries can take many shapes.

- If they are numbers, then the child process inherits that

same zero-indexed fd from the parent process.

- If uv_stream_t handles are passed in, those are used as

a read-write pipe or inherited stream depending if the
stream has a valid fd.

- Including nil placeholders means to ignore that fd in

the child process.

When the child process exits, on_exit is called with an exit
code and signal.

Returns: uv_process_t userdata or nil, integer or
string, uv.error_name or nil

uv.process_kill({process}, {signame}) uv.process_kill()

> method form process:kill(signame)

Parameters:

- process: uv_process_t userdata

- signame: string or integer or nil (default: sigterm)

Sends the specified signal to the given process handle. Check
the documentation on uv_signal_t for signal support,
specially on Windows.

See luv-constants for supported signame input and output
values.

Returns: 0 or fail

uv.kill({pid}, {signame}) uv.kill()

Parameters:

- pid: integer

- signame: string or integer or nil (default: sigterm)

Sends the specified signal to the given PID. Check the
documentation on uv_signal_t for signal support, specially
on Windows.

See luv-constants for supported signame input and output
values.

Returns: 0 or fail

uv.process_get_pid({process}) uv.process_get_pid()

> method form process:get_pid()

Parameters:

- process: uv_process_t userdata

Returns the handle's pid.

Returns: integer

uv_stream_t — Stream handle luv-stream-handle uv_stream_t

> uv_handle_t functions also apply.

Stream handles provide an abstraction of a duplex communication channel.
uv_stream_t is an abstract type, libuv provides 3 stream implementations
in the form of uv_tcp_t, uv_pipe_t and uv_tty_t.

uv.shutdown({stream} [, {callback}]) uv.shutdown()

> method form stream:shutdown([callback])

Parameters:

- stream: userdata for sub-type of uv_stream_t

- callback: callable or nil
    - err: nil or string

Shutdown the outgoing (write) side of a duplex stream. It waits for pending write requests to complete. The callback is called after shutdown is complete.

Returns: uv_shutdown_t userdata or fail

uv.listen({stream}, {backlog}, {callback}) uv.listen()

> method form stream:listen(backlog, callback)

Parameters:

- stream: userdata for sub-type of uv_stream_t
- backlog: integer
- callback: callable
    - err: nil or string

Start listening for incoming connections. backlog indicates the number of connections the kernel might queue, same as listen(2). When a new incoming connection is received the callback is called.

Returns: 0 or fail

uv.accept({stream}, {client_stream}) uv.accept()

> method form stream:accept(client_stream)

Parameters:

- stream: userdata for sub-type of uv_stream_t
- client_stream: userdata for sub-type of uv_stream_t

This call is used in conjunction with uv.listen() to accept incoming connections. Call this function after receiving a callback to accept the connection.

When the connection callback is called it is guaranteed that this function will complete successfully the first time. If you attempt to use it more than once, it may fail. It is suggested to only call this function once per connection call.

Returns: 0 or fail

```
server:listen(128, function (err)
  local client = uv.new_tcp()
  server:accept(client)
end)
```

uv.read_start({stream}, {callback}) uv.read_start()

> method form stream:read_start(callback)

Parameters:

- stream: userdata for sub-type of uv_stream_t
- callback: callable
    - err: nil or string
    - data: string or nil

Read data from an incoming stream. The callback will be made several times until there is no more data to read or uv.read_stop() is called. When we've reached EOF, data will be nil.

Returns: 0 or fail

```
stream:read_start(function (err, chunk)
  if err then
    -- handle read error
  elseif chunk then
    -- handle data
  else
    -- handle disconnect
  end
end)
```

uv.read_stop({stream}) uv.read_stop()

> method form stream:read_stop()

Parameters:

- stream: userdata for sub-type of uv_stream_t

Stop reading data from the stream. The read callback will no longer be called.

This function is idempotent and may be safely called on a stopped stream.

Returns: 0 or fail

uv.write({stream}, {data} [, {callback}]) uv.write()

> method form stream:write(data, [callback])

Parameters:

- stream: userdata for sub-type of uv_stream_t
- data: buffer
- callback: callable or nil
    - err: nil or string

Write data to stream.

data can either be a Lua string or a table of strings. If a table is passed in, the C backend will use writev to send all strings in a single system call.

The optional callback is for knowing when the write is complete.

Returns: uv_write_t userdata or fail

uv.write2({stream}, {data}, {send_handle} [, {callback}]) uv.write2()

> method form stream:write2(data, send_handle, [callback])

Parameters:

- stream: userdata for sub-type of uv_stream_t
- data: buffer
- send_handle: userdata for sub-type of uv_stream_t
- callback: callable or nil
    - err: nil or string

Extended write function for sending handles over a pipe. The pipe must be initialized with ipc option true.

Returns: uv_write_t userdata or fail

Note: send_handle must be a TCP socket or pipe, which is a server or a connection (listening or connected state). Bound sockets or pipes will be assumed to be servers.

uv.try_write({stream}, {data}) uv.try_write()

> method form stream:try_write(data)

Parameters:

- stream: userdata for sub-type of uv_stream_t
- data: buffer

Same as uv.write(), but won't queue a write request if it can't be completed immediately.

Will return number of bytes written (can be less than the supplied buffer size).

Returns: integer or fail

uv.try_write2({stream}, {data}, {send_handle}) uv.try_write2()

> method form stream:try_write2(data, send_handle)

Parameters:

- stream: userdata for sub-type of uv_stream_t
- data: buffer
- send_handle: userdata for sub-type of uv_stream_t

Like uv.write2(), but with the properties of uv.try_write(). Not supported on Windows, where it returns UV_EAGAIN.

Will return number of bytes written (can be less than the supplied buffer size).

Returns: integer or fail

uv.is_readable({stream}) uv.is_readable()

> method form stream:is_readable()

Parameters:

- stream: userdata for sub-type of uv_stream_t

Returns true if the stream is readable, false otherwise.

Returns: boolean

uv.is_writable({stream}) uv.is_writable()

> method form stream:is_writable()

Parameters:

- stream: userdata for sub-type of uv_stream_t

Returns true if the stream is writable, false otherwise.

Returns: boolean

uv.stream_set_blocking({stream}, {blocking}) uv.stream_set_blocking()

> method form stream:set_blocking(blocking)

Parameters:

- stream: userdata for sub-type of uv_stream_t
- blocking: boolean

Enable or disable blocking mode for a stream.

When blocking mode is enabled all writes complete synchronously. The interface remains unchanged otherwise, e.g. completion or failure of the operation will still be reported through a callback which is made asynchronously.

Returns: 0 or fail

WARNING: Relying too much on this API is not recommended. It is likely to change significantly in the future. Currently this only works on Windows and only for uv_pipe_t handles. Also libuv currently makes no ordering guarantee when the blocking mode is changed after write requests have already been submitted. Therefore it is recommended to set the blocking mode immediately after opening or creating the stream.

uv.stream_get_write_queue_size() uv.stream_get_write_queue_size()

> method form stream:get_write_queue_size()

Returns the stream's write queue size.

Returns: integer

uv_tcp_t – TCP handle luv-tcp-handle uv_tcp_t

> uv_handle_t and uv_stream_t functions also apply.

TCP handles are used to represent both TCP streams and servers.

uv.new_tcp([{flags}]) uv.new_tcp()

Parameters:

- flags: string or integer or nil

Creates and initializes a new uv_tcp_t. Returns the Lua userdata wrapping it.

If set, flags must be a valid address family. See luv-constants for supported address family input values.

Returns: uv_tcp_t userdata or fail

uv.tcp_open({tcp}, {sock}) uv.tcp_open()

> method form tcp:open(sock)

Parameters:

- tcp: uv_tcp_t userdata
- sock: integer

Open an existing file descriptor or SOCKET as a TCP handle.

Returns: 0 or fail

Note: The passed file descriptor or SOCKET is not checked for its type, but it's required that it represents a valid stream socket.

uv.tcp_nodelay({tcp}, {enable}) uv.tcp_nodelay()

> method form tcp:nodelay(enable)

Parameters:

- tcp: uv_tcp_t userdata
- enable: boolean

Enable / disable Nagle's algorithm.

Returns: 0 or fail

uv.tcp_keepalive()

uv.tcp_keepalive({tcp}, {enable} [, {delay}, {intvl}, {cnt}])

> method form tcp:keepalive(enable, [delay], [intvl], [cnt])

Parameters:

- tcp: uv_tcp_t userdata
- enable: boolean
- delay: integer or nil
- intvl: integer or nil
- cnt: integer or nil

Enable / disable TCP keep-alive. delay is the initial delay in seconds, intvl is the time in seconds between individual keep-alive probes, and cnt is the number of probes to send before assuming the connection is dead.
Ignored when enable is false.

Returns: 0 or fail

Note: intvl and cnt are only supported with Libuv >= 1.52.0.

uv.tcp_simultaneous_accepts({tcp}, {enable}) uv.tcp_simultaneous_accepts()

> method form tcp:simultaneous_accepts(enable)

Parameters:

- tcp: uv_tcp_t userdata
- enable: boolean

Enable / disable simultaneous asynchronous accept requests that are queued by the operating system when listening for new TCP connections.

This setting is used to tune a TCP server for the desired performance. Having simultaneous accepts can significantly improve the rate of accepting connections (which is why it is enabled by default) but may lead to uneven load distribution in multi-process setups.

Returns: 0 or fail

uv.tcp_bind({tcp}, {host}, {port} [, {flags}]) uv.tcp_bind()

> method form tcp:bind(host, port, [flags])

Parameters:

- tcp: uv_tcp_t userdata
- host: string

- port: integer
- flags: table or nil
    - ipv6only: boolean

Bind the handle to an host and port. host should be an IP address and not a domain name. Any flags are set with a table with field ipv6only equal to true or false.

When the port is already taken, you can expect to see an EADDRINUSE error from either uv.tcp_bind(), uv.listen() or uv.tcp_connect(). That is, a successful call to this function does not guarantee that the call to uv.listen() or uv.tcp_connect() will succeed as well.

Use a port of 0 to let the OS assign an ephemeral port. You can look it up later using uv.tcp_getsockname().

Returns: 0 or fail

uv.tcp_getpeername({tcp}) uv.tcp_getpeername()

> method form tcp:getpeername()

Parameters:
- tcp: uv_tcp_t userdata

Get the address of the peer connected to the handle.

See luv-constants for supported address family output values.

Returns: table or fail
- ip : string
- family : string
- port : integer

uv.tcp_getsockname({tcp}) uv.tcp_getsockname()

> method form tcp:getsockname()

Parameters:
- tcp: uv_tcp_t userdata

Get the current address to which the handle is bound.

See luv-constants for supported address family output values.

Returns: table or fail
- ip : string
- family : string
- port : integer

uv.tcp_connect({tcp}, {host}, {port}, {callback}) uv.tcp_connect()

> method form tcp:connect(host, port, callback)

Parameters:
- tcp: uv_tcp_t userdata
- host: string
- port: integer
- callback: callable
    - err: nil or string

Establish an IPv4 or IPv6 TCP connection.

Returns: uv_connect_t userdata or fail

```
local client = uv.new_tcp()
client:connect("127.0.0.1", 8080, function (err)
  -- check error and carry on.
end)
```

uv.tcp_write_queue_size({tcp}) uv.tcp_write_queue_size()

> method form tcp:write_queue_size()

DEPRECATED: Please use uv.stream_get_write_queue_size() instead.

uv.tcp_close_reset([{callback}]) uv.tcp_close_reset()

> method form tcp:close_reset([callback])

Parameters:

- tcp: uv_tcp_t userdata
- callback: callable or nil

Resets a TCP connection by sending a RST packet. This is accomplished by setting the SO_LINGER socket option with a linger interval of zero and then calling uv.close(). Due to some platform inconsistencies, mixing of uv.shutdown() and uv.tcp_close_reset() calls is not allowed.

Returns: 0 or fail

uv.socketpair()

uv.socketpair([{socktype}, [{protocol}, [{flags1}, [{flags2}]]]])

Parameters:

- socktype: string, integer or nil (default: stream)
- protocol: string, integer or nil (default: 0)
- flags1: table or nil
  - nonblock: boolean (default: false)
- flags2: table or nil
  - nonblock: boolean (default: false)

Create a pair of connected sockets with the specified properties. The resulting handles can be passed to uv.tcp_open(), used with uv.spawn(), or for any other purpose.

See luv-constants for supported socktype input values.

When protocol is set to 0 or nil, it will be automatically chosen based on the socket's domain and type. When protocol is specified as a string, it will be looked up using the getprotobyname(3) function (examples: "ip", "icmp", "tcp", "udp", etc).

Flags:

- nonblock: Opens the specified socket handle for OVERLAPPED or FIONBIO/`O_NONBLOCK` I/O usage. This is recommended for handles that will be used by libuv, and not usually recommended otherwise.

Equivalent to socketpair(2) with a domain of AF_UNIX.

Returns: table or fail

- [1, 2] : integer (file descriptor)

```
-- Simple read/write with tcp
local fds = uv.socketpair(nil, nil, {nonblock=true}, {nonblock=true})

local sock1 = uv.new_tcp()
sock1:open(fds[1])

local sock2 = uv.new_tcp()
sock2:open(fds[2])

sock1:write("hello")
sock2:read_start(function(err, chunk)
  assert(not err, err)
  print(chunk)
end)
```

uv_pipe_t — Pipe handle

luv-pipe-handle uv_pipe_t

> uv_handle_t and uv_stream_t functions also apply.

Pipe handles provide an abstraction over local domain sockets on Unix and named pipes on Windows.

```
local pipe = uv.new_pipe(false)

pipe:bind('/tmp/sock.test')

pipe:listen(128, function()
  local client = uv.new_pipe(false)
  pipe:accept(client)
  client:write("hello!\n")
  client:close()
end)
```

uv.new_pipe([{ipc}]) uv.new_pipe()

Parameters:

- ipc: boolean or nil (default: false)

Creates and initializes a new uv_pipe_t. Returns the Lua userdata wrapping it. The ipc argument is a boolean to indicate if this pipe will be used for handle passing between processes.

Returns: uv_pipe_t userdata or fail

uv.pipe_open({pipe}, {fd}) uv.pipe_open()

> method form pipe:open(fd)

Parameters:

- pipe: uv_pipe_t userdata
- fd: integer

Open an existing file descriptor or uv_handle_t as a pipe.

Returns: 0 or fail

Note: The file descriptor is set to non-blocking mode.

uv.pipe_bind({pipe}, {name}) uv.pipe_bind()

> method form pipe:bind(name)

Parameters:

- pipe: uv_pipe_t userdata
- name: string

Bind the pipe to a file path (Unix) or a name (Windows).

Returns: 0 or fail

Note: Paths on Unix get truncated to sizeof(sockaddr_un.sun_path) bytes, typically between 92 and 108 bytes.

uv.pipe_connect({pipe}, {name} [, {callback}]) uv.pipe_connect()

> method form pipe:connect(name, [callback])

Parameters:

- pipe: uv_pipe_t userdata
- name: string
- callback: callable or nil
    - err: nil or string

Connect to the Unix domain socket or the named pipe.

Returns: uv_connect_t userdata or fail

Note: Paths on Unix get truncated to sizeof(sockaddr_un.sun_path) bytes, typically between 92 and 108 bytes.

uv.pipe_getsockname({pipe}) uv.pipe_getsockname()

> method form pipe:getsockname()

Parameters:

- pipe: uv_pipe_t userdata

Get the name of the Unix domain socket or the named pipe.

Returns: string or fail

uv.pipe_getpeername({pipe}) uv.pipe_getpeername()

> method form pipe:getpeername()

Parameters:

- pipe: uv_pipe_t userdata

Get the name of the Unix domain socket or the named pipe to which the handle is connected.

Returns: string or fail

uv.pipe_pending_instances({pipe}, {count}) uv.pipe_pending_instances()

> method form pipe:pending_instances(count)

Parameters:

- pipe: uv_pipe_t userdata
- count: integer

Set the number of pending pipe instance handles when the pipe server is waiting for connections.

Returns: Nothing.

Note: This setting applies to Windows only.

uv.pipe_pending_count({pipe}) uv.pipe_pending_count()

> method form pipe:pending_count()

Parameters:

- pipe: uv_pipe_t userdata

Returns the pending pipe count for the named pipe.

Returns: integer

uv.pipe_pending_type({pipe}) uv.pipe_pending_type()

> method form pipe:pending_type()

Parameters:

- pipe: uv_pipe_t userdata

Used to receive handles over IPC pipes.

First - call uv.pipe_pending_count(), if it's > 0 then initialize a handle of the given type, returned by uv.pipe_pending_type() and call uv.accept(pipe, handle) .

Returns: string

uv.pipe_chmod({pipe}, {flags}) uv.pipe_chmod()

> method form pipe:chmod(flags)

Parameters:

- pipe: uv_pipe_t userdata
- flags: string

Alters pipe permissions, allowing it to be accessed from processes run by different users. Makes the pipe writable or readable by all users. flags are: "r", "w", "rw", or "wr" where r is READABLE and w is WRITABLE. This function is blocking.

Returns: 0 or fail

uv.pipe({read_flags}, {write_flags}) uv.pipe()

Parameters:

- read_flags: table or nil
  - nonblock: boolean (default: false)
- write_flags: table or nil
  - nonblock: boolean (default: false)

Create a pair of connected pipe handles. Data may be written to the write fd and read from the read fd. The resulting

handles can be passed to pipe_open, used with spawn, or
for any other purpose.

Flags:

- nonblock: Opens the specified socket handle for

  OVERLAPPED or FIONBIO/`O_NONBLOCK` I/O usage. This is
  recommended for handles that will be used by libuv, and not
  usually recommended otherwise.

Equivalent to pipe(2) with the O_CLOEXEC flag set.

Returns: table or fail

- read : integer (file descriptor)

- write : integer (file descriptor)

```
-- Simple read/write with pipe_open
local fds = uv.pipe({nonblock=true}, {nonblock=true})

local read_pipe = uv.new_pipe()
read_pipe:open(fds.read)

local write_pipe = uv.new_pipe()
write_pipe:open(fds.write)

write_pipe:write("hello")
read_pipe:read_start(function(err, chunk)
  assert(not err, err)
  print(chunk)
end)
```

uv.pipe_bind2({pipe}, {name}, {flags}) uv.pipe_bind2()

> method form pipe:pipe_bind(name, flags)

Parameters:

- pipe: uv_pipe_t userdata

- name: string

- flags: integer or table or nil (default: 0)

Flags:

- If type(flags) is number, it must be 0 or

  uv.constants.PIPE_NO_TRUNCATE.
- If type(flags) is table, it must be {} or

  { no_truncate = true|false }.
- If type(flags) is nil, it use default value 0.

- Returns EINVAL for unsupported flags without performing the

  bind.

Bind the pipe to a file path (Unix) or a name (Windows).

Supports Linux abstract namespace sockets. namelen must include
the leading '\0' byte but not the trailing nul byte.

Returns: 0 or fail

Note:

1. Paths on Unix get truncated to sizeof(sockaddr_un.sun_path)

   bytes, typically between 92 and 108 bytes.
2. New in version 1.46.0.

uv.pipe_connect2(pipe, name, [flags], [callback]) uv.pipe_connect2()

> method form pipe:connect2(name, [flags], [callback])

Parameters:

- pipe: uv_pipe_t userdata

- name: string

- flags: integer or table or nil (default: 0)

- callback: callable or nil
    - err: nil or string

Flags:

- If type(flags) is number, it must be 0 or uv.constants.PIPE_NO_TRUNCATE.
- If type(flags) is table, it must be {} or { no_truncate = true|false }.
- If type(flags) is nil, it use default value 0.
- Returns EINVAL for unsupported flags without performing the bind operation.

Connect to the Unix domain socket or the named pipe.

Supports Linux abstract namespace sockets. namelen must include the leading nul byte but not the trailing nul byte.

Returns: uv_connect_t userdata or fail

Note:

1. Paths on Unix get truncated to sizeof(sockaddr_un.sun_path) bytes, typically between 92 and 108 bytes.
2. New in version 1.46.0.

uv_tty_t — TTY handle

luv-tty-handle uv_tty_t

> uv_handle_t and uv_stream_t functions also apply.

TTY handles represent a stream for the console.

```
-- Simple echo program
local stdin = uv.new_tty(0, true)
local stdout = uv.new_tty(1, false)

stdin:read_start(function (err, data)
  assert(not err, err)
  if data then
    stdout:write(data)
  else
    stdin:close()
    stdout:close()
  end
end)
```

uv.new_tty({fd}, {readable}) uv.new_tty()

Parameters:

- fd: integer
- readable: boolean

Initialize a new TTY stream with the given file descriptor. Usually the file descriptor will be:

- 0 - stdin
- 1 - stdout
- 2 - stderr

On Unix this function will determine the path of the fd of the terminal using ttyname_r(3), open it, and use it if the passed file descriptor refers to a TTY. This lets libuv put the tty in non-blocking mode without affecting other processes that share the tty.

This function is not thread safe on systems that don’t support ioctl TIOCGPTN or TIOCPTYGNAME, for instance OpenBSD and Solaris.

Returns: uv_tty_t userdata or fail

Note: If reopening the TTY fails, libuv falls back to blocking writes.

uv.tty_set_mode({tty}, {mode}) uv.tty_set_mode()

> method form tty:set_mode(mode)

Parameters:

- tty: uv_tty_t userdata
- mode: string or integer

Set the TTY using the specified terminal mode.

See luv-constants for supported TTY mode input values.

Returns: 0 or fail

uv.tty_reset_mode() uv.tty_reset_mode()

To be called when the program exits. Resets TTY settings to default values for the next process to take over.

This function is async signal-safe on Unix platforms but can fail with error code EBUSY if you call it when execution is inside uv.tty_set_mode().

Returns: 0 or fail

uv.tty_get_winsize({tty}) uv.tty_get_winsize()

> method form tty:get_winsize()

Parameters:

- tty: uv_tty_t userdata

Gets the current Window width and height.

Returns: integer, integer or fail

uv.tty_set_vterm_state({state}) uv.tty_set_vterm_state()

Parameters:

- state: string

Controls whether console virtual terminal sequences are processed by libuv or console. Useful in particular for enabling ConEmu support of ANSI X3.64 and Xterm 256 colors. Otherwise Windows10 consoles are usually detected automatically. State should be one of: "supported" or "unsupported".

This function is only meaningful on Windows systems. On Unix it is silently ignored.

Returns: none

uv.tty_get_vterm_state() uv.tty_get_vterm_state()

Get the current state of whether console virtual terminal sequences are handled by libuv or the console. The return value is "supported" or "unsupported".

This function is not implemented on Unix, where it returns ENOTSUP.

Returns: string or fail

uv_udp_t — UDP handle luv-udp-handle uv_udp_t

> uv_handle_t functions also apply.

UDP handles encapsulate UDP communication for both clients and servers.

uv.new_udp([{flags}]) uv.new_udp()

Parameters:

- flags: table or nil
  - family: string or nil
  - mmsgs: integer or nil (default: 1)

Creates and initializes a new uv_udp_t. Returns the Lua userdata wrapping it. The actual socket is created lazily.

See luv-constants for supported address family input values.

When specified, mmsgs determines the number of messages able to be received at one time via recvmmsg(2) (the allocated buffer will be sized to be able to fit the specified number of max size dgrams). Only has an effect on platforms that support recvmmsg(2).

Note: For backwards compatibility reasons, flags can also be a string or integer. When it is a string, it will be treated

like the family key above. When it is an integer, it will be used directly as the flags parameter when calling uv_udp_init_ex.

Returns: uv_udp_t userdata or fail

uv.udp_get_send_queue_size() uv.udp_get_send_queue_size()

> method form udp:get_send_queue_size()

Returns the handle's send queue size.

Returns: integer

uv.udp_get_send_queue_count() uv.udp_get_send_queue_count()

> method form udp:get_send_queue_count()

Returns the handle's send queue count.

Returns: integer

uv.udp_open({udp}, {fd}, [{flags}]) uv.udp_open()

> method form udp:open(fd, [flags])

Parameters:

- udp: uv_udp_t userdata
- fd: integer
- flags: table or integer or nil
  - reuseaddr: boolean or nil
  - reuseport: boolean or nil

Opens an existing file descriptor or Windows SOCKET as a UDP handle.

Unix only: The only requirement of the sock argument is that it follows the datagram contract (works in unconnected mode, supports sendmsg()/recvmsg(), etc). In other words, other datagram-type sockets like raw sockets or netlink sockets can also be passed to this function.

The file descriptor is set to non-blocking mode.

Note: The passed file descriptor or SOCKET is not checked for its type, but it's required that it represents a valid datagram socket.

Returns: 0 or fail

Note: flags is only supported with Libuv >= 1.52.0.

uv.udp_bind({udp}, {host}, {port} [, {flags}]) uv.udp_bind()

> method form udp:bind(host, port, [flags])

Parameters:

- udp: uv_udp_t userdata
- host: string
- port: number
- flags: table or nil
  - ipv6only: boolean or nil
  - reuseaddr: boolean or nil
  - linux_recverr: boolean or nil
  - reuseport: boolean or nil

Bind the UDP handle to an IP address and port. Any flags are set with a table with fields reuseaddr, ipv6only, linux_recverr, reuseport equal to true or false.

- reuseaddr: Indicates if SO_REUSEADDR will be set when

  binding the handle. This sets the SO_REUSEPORT socket flag on the BSDs (except for DragonFlyBSD), OS X, and other platforms where SO_REUSEPORTs don't have the capability of load balancing, as the opposite of what reuseport would do. On

other Unix platforms, it sets the SO_REUSEADDR flag. What that means is that multiple threads or processes can bind to the same address without error (provided they all set the flag) but only the last one to bind will receive any traffic, in effect "stealing" the port from the previous listener.

- ipv6only: Disables dual stack mode.

- linux_recverr: Indicates if IP_RECVERR/IPV6_RECVERR will

be set when binding the handle. This sets IP_RECVERR for IPv4 and IPV6_RECVERR for IPv6 UDP sockets on Linux. This stops the Linux kernel from suppressing some ICMP error messages and enables full ICMP error reporting for faster failover. This flag is no-op on platforms other than Linux.

- reuseport: Indicates if SO_REUSEPORT will be set when

binding the handle. This sets the SO_REUSEPORT socket option on supported platforms. Unlike reuseaddr, this flag will make multiple threads or processes that are binding to the same address and port "share" the port, which means incoming datagrams are distributed across the receiving sockets among threads or processes. This flag is available only on Linux 3.9+, DragonFlyBSD 3.6+, FreeBSD 12.0+, Solaris 11.4, and AIX 7.2.5+ for now.

Returns: 0 or fail

Note: The flag linux_recverr is only supported with Libuv >= 1.42.0. The flag reuseport is only supported with Libuv >= 1.49.0.

uv.udp_getsockname({udp}) uv.udp_getsockname()

> method form udp:getsockname()

Parameters:

- udp: uv_udp_t userdata

Get the local IP and port of the UDP handle.

Returns: table or fail

- ip : string

- family : string

- port : integer

uv.udp_getpeername({udp}) uv.udp_getpeername()

> method form udp:getpeername()

Parameters:

- udp: uv_udp_t userdata

Get the remote IP and port of the UDP handle on connected UDP handles.

Returns: table or fail

- ip : string

- family : string

- port : integer

uv.udp_set_membership()

uv.udp_set_membership({udp}, {multicast_addr}, {interface_addr}, {membership})

> method form
> udp:set_membership(multicast_addr, interface_addr, membership)

Parameters:

- udp: uv_udp_t userdata

- multicast_addr: string

- interface_addr: string or nil

- membership: string

Set membership for a multicast address. multicast_addr is
multicast address to set membership for. interface_addr is
interface address. membership can be the string "leave" or
"join".

Returns: 0 or fail

uv.udp_set_source_membership()

uv.udp_set_source_membership({udp}, {multicast_addr}, {interface_addr}, {source_addr}, {membership})

> method form
> udp:set_source_membership(multicast_addr, interface_addr, source_addr, membership)

Parameters:

- udp: uv_udp_t userdata
- multicast_addr: string
- interface_addr: string or nil
- source_addr: string
- membership: string

Set membership for a source-specific multicast group.
multicast_addr is multicast address to set membership for.
interface_addr is interface address. source_addr is source
address. membership can be the string "leave" or "join".

Returns: 0 or fail

uv.udp_set_multicast_loop({udp}, {on}) uv.udp_set_multicast_loop()

> method form udp:set_multicast_loop(on)

Parameters:

- udp: uv_udp_t userdata
- on: boolean

Set IP multicast loop flag. Makes multicast packets loop back
to local sockets.

Returns: 0 or fail

uv.udp_set_multicast_ttl({udp}, {ttl}) uv.udp_set_multicast_ttl()

> method form udp:set_multicast_ttl(ttl)

Parameters:

- udp: uv_udp_t userdata
- ttl: integer

Set the multicast ttl.

ttl is an integer 1 through 255.

Returns: 0 or fail

uv.udp_set_multicast_interface()

uv.udp_set_multicast_interface({udp}, {interface_addr})

> method form udp:set_multicast_interface(interface_addr)

Parameters:

- udp: uv_udp_t userdata
- interface_addr: string

Set the multicast interface to send or receive data on.

Returns: 0 or fail

uv.udp_set_broadcast({udp}, {on}) uv.udp_set_broadcast()

> method form udp:set_broadcast(on)

Parameters:

- udp: uv_udp_t userdata
- on: boolean

Set broadcast on or off.

Returns: 0 or fail

uv.udp_set_ttl({udp}, {ttl}) uv.udp_set_ttl()

> method form udp:set_ttl(ttl)

Parameters:

- udp: uv_udp_t userdata
- ttl: integer

Set the time to live.

ttl is an integer 1 through 255.

Returns: 0 or fail

uv.udp_send({udp}, {data}, {host}, {port}, {callback}) uv.udp_send()

> method form udp:send(data, host, port, callback)

Parameters:

- udp: uv_udp_t userdata
- data: buffer
- host: string
- port: integer
- callback: callable
  - err: nil or string

Send data over the UDP socket. If the socket has not previously been bound with uv.udp_bind() it will be bound to 0.0.0.0 (the "all interfaces" IPv4 address) and a random port number.

Returns: uv_udp_send_t userdata or fail

uv.udp_try_send({udp}, {data}, {host}, {port}) uv.udp_try_send()

> method form udp:try_send(data, host, port)

Parameters:

- udp: uv_udp_t userdata
- data: buffer
- host: string
- port: integer

Same as uv.udp_send(), but won't queue a send request if it can't be completed immediately.

Returns: integer or fail

uv.udp_try_send2({udp}, {messages}, {flags}) uv.udp_try_send2()

> method form udp:try_send2(messages, flags)

Parameters:

- udp: uv_udp_t userdata
- messages: table
  - [1, 2, 3, ..., n] : table
    - data : buffer
    - addr : table
      - ip : string
      - port : integer

- flags: nil (see below)
- port: integer

Like uv.udp_try_send(), but can send multiple datagrams.
Lightweight abstraction around sendmmsg(2), with a
sendmsg(2) fallback loop for platforms that do not support
the former. The udp handle must be fully initialized, either
from a uv.udp_bind call, another call that will bind
automatically (udp_send, udp_try_send, etc), or from
uv.udp_connect. messages should be an array-like table,
where addr must be specified if the udp has not been
connected via udp_connect. Otherwise, addr must be nil.

flags is reserved for future extension and must currently be
nil or 0 or {}.

Returns the number of messages sent successfully. An error will only be returned
if the first datagram failed to be sent.

Returns: integer or fail

```
-- If client:connect(...) was not called
local addr = { ip = "127.0.0.1", port = 1234 }
client:try_send2({
  { data = "Message 1", addr = addr },
  { data = "Message 2", addr = addr },
})
-- If client:connect(...) was called
client:try_send2({
  { data = "Message 1" },
  { data = "Message 2" },
})
```

uv.udp_recv_start({udp}, {callback}) uv.udp_recv_start()

> method form udp:recv_start(callback)

Parameters:
- udp: uv_udp_t userdata
- callback: callable
  - err: nil or string
  - data: string or nil
  - addr: table or nil
    - ip: string
    - port: integer
    - family: string
  - flags: table
    - partial: boolean or nil
    - mmsg_chunk: boolean or nil

Prepare for receiving data. If the socket has not previously
been bound with uv.udp_bind() it is bound to 0.0.0.0 (the
"all interfaces" IPv4 address) and a random port number.

See luv-constants for supported address family output values.

Returns: 0 or fail

uv.udp_recv_stop({udp}) uv.udp_recv_stop()

> method form udp:recv_stop()

Parameters:
- udp: uv_udp_t userdata

Stop listening for incoming datagrams.

Returns: 0 or fail

uv.udp_connect({udp}, {host}, {port}) uv.udp_connect()

> method form udp:connect(host, port)

Parameters:

- udp: uv_udp_t userdata
- host: string
- port: integer

Associate the UDP handle to a remote address and port, so every message sent by this handle is automatically sent to that destination. Calling this function with a NULL addr disconnects the handle. Trying to call uv.udp_connect() on an already connected handle will result in an EISCONN error. Trying to disconnect a handle that is not connected will return an ENOTCONN error.

Returns: 0 or fail

uv_fs_event_t – FS Event handle luv-fs-event-handle uv_fs_event_t

> uv_handle_t functions also apply.

FS Event handles allow the user to monitor a given path for changes, for example, if the file was renamed or there was a generic change in it. This handle uses the best backend for the job on each platform.

uv.new_fs_event() uv.new_fs_event()

Creates and initializes a new uv_fs_event_t. Returns the Lua userdata wrapping it.

Returns: uv_fs_event_t userdata or fail

uv.fs_event_start({fs_event}, {path}, {flags}, {callback}) uv.fs_event_start()

> method form fs_event:start(path, flags, callback)

Parameters:

- fs_event: uv_fs_event_t userdata
- path: string
- flags: table
  - watch_entry: boolean or nil (default: false)
  - stat: boolean or nil (default: false)
  - recursive: boolean or nil (default: false)
- callback: callable
  - err: nil or string
  - filename: string
  - events: table
    - change: boolean or nil
    - rename: boolean or nil

Start the handle with the given callback, which will watch the specified path for changes.

Returns: 0 or fail

uv.fs_event_stop() uv.fs_event_stop()

> method form fs_event:stop()

Stop the handle, the callback will no longer be called.

Returns: 0 or fail

uv.fs_event_getpath() uv.fs_event_getpath()

> method form fs_event:getpath()

Get the path being monitored by the handle.

Returns: string or fail

uv_fs_poll_t — FS Poll handle

luv-fs-poll-handle uv_fs_poll_t

> uv_handle_t functions also apply.

FS Poll handles allow the user to monitor a given path for changes. Unlike uv_fs_event_t, fs poll handles use stat to detect when a file has changed so they can work on file systems where fs event handles can't.

uv.new_fs_poll() uv.new_fs_poll()

Creates and initializes a new uv_fs_poll_t. Returns the Lua userdata wrapping it.

Returns: uv_fs_poll_t userdata or fail

uv.fs_poll_start({fs_poll}, {path}, {interval}, {callback}) uv.fs_poll_start()

> method form fs_poll:start(path, interval, callback)

Parameters:

- fs_poll: uv_fs_poll_t userdata
- path: string
- interval: integer
- callback: callable
  - err: nil or string
  - prev: table or nil (see uv.fs_stat)
  - curr: table or nil (see uv.fs_stat)

Check the file at path for changes every interval milliseconds.

Note: For maximum portability, use multi-second intervals. Sub-second intervals will not detect all changes on many file systems.

Returns: 0 or fail

uv.fs_poll_stop() uv.fs_poll_stop()

> method form fs_poll:stop()

Stop the handle, the callback will no longer be called.

Returns: 0 or fail

uv.fs_poll_getpath() uv.fs_poll_getpath()

> method form fs_poll:getpath()

Get the path being monitored by the handle.

Returns: string or fail

FILE SYSTEM OPERATIONS

luv-file-system-operations uv_fs_t

Most file system functions can operate synchronously or asynchronously. When a synchronous version is called (by omitting a callback), the function will immediately return the results of the FS call. When an asynchronous version is called (by providing a callback), the function will immediately return a uv_fs_t userdata and asynchronously execute its callback; if an error is encountered, the first and only argument passed to the callback will be the err error string; if the operation completes successfully, the first argument will be nil and the remaining arguments will be the results of the FS call.

Synchronous and asynchronous versions of readFile (with naive error handling) are implemented below as an example:

```
local function readFileSync(path)
  local fd = assert(uv.fs_open(path, "r", 438))
  local stat = assert(uv.fs_fstat(fd))
  local data = assert(uv.fs_read(fd, stat.size, 0))
  assert(uv.fs_close(fd))
  return data
end

local data = readFileSync("main.lua")
print("synchronous read", data)
```

```lua
local function readFile(path, callback)
  uv.fs_open(path, "r", 438, function(err, fd)
    assert(not err, err)
    uv.fs_fstat(fd, function(err, stat)
      assert(not err, err)
      uv.fs_read(fd, stat.size, 0, function(err, data)
        assert(not err, err)
        uv.fs_close(fd, function(err)
          assert(not err, err)
          return callback(data)
        end)
      end)
    end)
  end)
end

readFile("main.lua", function(data)
  print("asynchronous read", data)
end)
```

uv.fs_close({fd} [, {callback}]) uv.fs_close()

Parameters:

- fd: integer
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - success: boolean or nil

Equivalent to close(2).

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_open({path}, {flags}, {mode} [, {callback}]) uv.fs_open()

Parameters:

- path: string
- flags: string or integer
- mode: integer (octal chmod(1) mode, e.g. tonumber('644', 8))
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - fd: integer or nil

Equivalent to open(2). Access flags may be an integer or one of: "r", "rs", "sr", "r+", "rs+", "sr+", "w", "wx", "xw", "w+", "wx+", "xw+", "a", "ax", "xa", "a+", "ax+", or "`xa+`".

Returns (sync version): integer or fail

Returns (async version): uv_fs_t userdata

Note: On Windows, libuv uses CreateFileW and thus the file is always opened in binary mode. Because of this, the O_BINARY and O_TEXT flags are not supported.

uv.fs_read({fd}, {size} [, {offset} [, {callback}]]) uv.fs_read()

Parameters:

- fd: integer
- size: integer
- offset: integer or nil
- callback: callable (async version) or nil (sync version)

- err: nil or string
- data: string or nil

Equivalent to preadv(2). Returns any data. An empty string indicates EOF.

If offset is nil or omitted, it will default to -1, which indicates "use and update the current file offset."

Note: When offset is >= 0, the current file offset will not be updated by the read.

Returns (sync version): string or fail

Returns (async version): uv_fs_t userdata

uv.fs_unlink({path} [, {callback}]) uv.fs_unlink()

Parameters:

- path: string
- callback: callable (async version) or nil (sync version)
- err: nil or string
- success: boolean or nil

Equivalent to unlink(2).

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_write({fd}, {data} [, {offset} [, {callback}]]) uv.fs_write()

Parameters:

- fd: integer
- data: buffer
- offset: integer or nil
- callback: callable (async version) or nil (sync version)
- err: nil or string
- bytes: integer or nil

Equivalent to pwritev(2). Returns the number of bytes written.

If offset is nil or omitted, it will default to -1, which indicates "use and update the current file offset."

Note: When offset is >= 0, the current file offset will not be updated by the write.

Returns (sync version): integer or fail

Returns (async version): uv_fs_t userdata

uv.fs_mkdir({path}, {mode} [, {callback}]) uv.fs_mkdir()

Parameters:

- path: string
- mode: integer (octal representation of chmod(1) mode, e.g. tonumber('755', 8))
- callback: callable (async version) or nil (sync version)
- err: nil or string
- success: boolean or nil

Equivalent to mkdir(2).

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_mkdtemp({template} [, {callback}]) uv.fs_mkdtemp()

Parameters:

- template: string
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - path: string or nil

Equivalent to mkdtemp(3).

Returns (sync version): string or fail

Returns (async version): uv_fs_t userdata

uv.fs_mkstemp({template} [, {callback}]) uv.fs_mkstemp()

Parameters:

- template: string
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - fd: integer or nil
  - path: string or nil

Equivalent to mkstemp(3). Returns a temporary file handle and filename.

Returns (sync version): integer, string or fail

Returns (async version): uv_fs_t userdata

uv.fs_rmdir({path} [, {callback}]) uv.fs_rmdir()

Parameters:

- path: string
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - success: boolean or nil

Equivalent to rmdir(2).

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_scandir({path} [, {callback}]) uv.fs_scandir()

Parameters:

- path: string
- callback: callable
  - err: nil or string
  - success: uv_fs_t userdata or nil

Equivalent to scandir(3), with a slightly different API. Returns a handle that the user can pass to uv.fs_scandir_next().

Note: This function can be used synchronously or asynchronously. The request userdata is always synchronously returned regardless of whether a callback is provided and the same userdata is passed to the callback if it is provided.

Returns: uv_fs_t userdata or fail

uv.fs_scandir_next({fs}) uv.fs_scandir_next()

Parameters:

- fs: uv_fs_t userdata

Called on a uv_fs_t returned by uv.fs_scandir() to get the next directory entry data as a name, type pair. When there are no more entries, nil is returned.

Note: This function only has a synchronous version. See uv.fs_opendir() and its related functions for an asynchronous version.

Returns: string, string or nil or fail

uv.fs_stat({path} [, {callback}]) uv.fs_stat()

Parameters:

- path: string
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - stat: table or nil (see below)

Equivalent to stat(2).

Returns (sync version): table or fail

- dev : integer
- mode : integer
- nlink : integer
- uid : integer
- gid : integer
- rdev : integer
- ino : integer
- size : integer
- blksize : integer
- blocks : integer
- flags : integer
- gen : integer
- atime : table
  - sec : integer
  - nsec : integer
- mtime : table
  - sec : integer
  - nsec : integer
- ctime : table

- sec : integer
- nsec : integer

- birthtime : table
  - sec : integer
  - nsec : integer

- type : string

Returns (async version): uv_fs_t userdata

uv.fs_fstat({fd} [, {callback}]) uv.fs_fstat()

Parameters:

- fd: integer
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - stat: table or nil (see uv.fs_stat)

Equivalent to fstat(2).

Returns (sync version): table or fail (see uv.fs_stat)

Returns (async version): uv_fs_t userdata

uv.fs_lstat({path} [, {callback}]) uv.fs_lstat()

Parameters:

- path: string
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - stat: table or nil (see uv.fs_stat)

Equivalent to lstat(2).

Returns (sync version): table or fail (see uv.fs_stat())

Returns (async version): uv_fs_t userdata

uv.fs_rename({path}, {new_path} [, {callback}]) uv.fs_rename()

Parameters:

- path: string
- new_path: string
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - success: boolean or nil

Equivalent to rename(2).

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_fsync({fd} [, {callback}]) uv.fs_fsync()

Parameters:

- fd: integer

- callback: callable (async version) or nil (sync version)
    - err: nil or string
    - success: boolean or nil

  Equivalent to fsync(2).

  Returns (sync version): boolean or fail

  Returns (async version): uv_fs_t userdata

uv.fs_fdatasync({fd} [, {callback}]) uv.fs_fdatasync()

  Parameters:

- fd: integer
- callback: callable (async version) or nil (sync version)
    - err: nil or string
    - success: boolean or nil

  Equivalent to fdatasync(2).

  Returns (sync version): boolean or fail

  Returns (async version): uv_fs_t userdata

uv.fs_ftruncate({fd}, {offset} [, {callback}]) uv.fs_ftruncate()

  Parameters:

- fd: integer
- offset: integer
- callback: callable (async version) or nil (sync version)
    - err: nil or string
    - success: boolean or nil

  Equivalent to ftruncate(2).

  Returns (sync version): boolean or fail

  Returns (async version): uv_fs_t userdata

uv.fs_sendfile() uv.fs_sendfile()

uv.fs_sendfile({out_fd}, {in_fd}, {in_offset}, {size} [, {callback}])

  Parameters:

- out_fd: integer
- in_fd: integer
- in_offset: integer
- size: integer
- callback: callable (async version) or nil (sync version)
    - err: nil or string
    - bytes: integer or nil

  Limited equivalent to sendfile(2). Returns the number of bytes written.

  Returns (sync version): integer or fail

  Returns (async version): uv_fs_t userdata

uv.fs_access({path}, {mode} [, {callback}]) uv.fs_access()

Parameters:

- path: string
- mode: integer string (a combination of the 'r', 'w' and 'x' characters denoting the symbolic mode as per chmod(1))
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - permission: boolean or nil

Equivalent to access(2) on Unix. Windows uses GetFileAttributesW(). Access mode can be an integer or a string containing "R" or "W" or "X". Returns true or false indicating access permission.

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_chmod({path}, {mode} [, {callback}]) uv.fs_chmod()

Parameters:

- path: string
- mode: integer (octal representation of chmod(1) mode, e.g. tonumber('644', 8))
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - success: boolean or nil

Equivalent to chmod(2).

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_fchmod({fd}, {mode} [, {callback}]) uv.fs_fchmod()

Parameters:

- fd: integer
- mode: integer
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - success: boolean or nil

Equivalent to fchmod(2).

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_utime({path} [, {atime}, {mtime}, {callback}]) uv.fs_utime()

Parameters:

- path: string
- atime: number or string or nil
- mtime: number or string or nil
- callback: callable (async version) or nil (sync

version)
- err: nil or string
- success: boolean or nil

Equivalent to utime(2).

See luv-constants for supported FS Modification Time constants.

Passing "now" or uv.constants.FS_UTIME_NOW as the atime or mtime sets the timestamp to the current time.

Passing nil, "omit", or uv.constants.FS_UTIME_OMIT as the atime or mtime leaves the timestamp untouched.

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_futime({fd} [, {atime}, {mtime}, {callback}]) uv.fs_futime()

Parameters:
- fd: integer
- atime: number or string or nil
- mtime: number or string or nil
- callback: callable (async version) or nil (sync version)
- err: nil or string
- success: boolean or nil

Equivalent to futimes(3).

See luv-constants for supported FS Modification Time constants.

Passing "now" or uv.constants.FS_UTIME_NOW as the atime or mtime sets the timestamp to the current time.

Passing nil, "omit", or uv.constants.FS_UTIME_OMIT as the atime or mtime leaves the timestamp untouched.

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_lutime({path} [, {atime}, {mtime}, {callback}]) uv.fs_lutime()

Parameters:
- path: string
- atime: number or string or nil
- mtime: number or string or nil
- callback: callable (async version) or nil (sync version)
- err: nil or string
- success: boolean or nil

Equivalent to lutimes(3).

See luv-constants for supported FS Modification Time constants.

Passing "now" or uv.constants.FS_UTIME_NOW as the atime or mtime sets the timestamp to the current time.

Passing nil, "omit", or uv.constants.FS_UTIME_OMIT as the atime or mtime leaves the timestamp untouched.

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_link({path}, {new_path} [, {callback}]) uv.fs_link()

Parameters:

- path: string
- new_path: string
- callback: callable (async version) or nil (sync version)
    - err: nil or string
    - success: boolean or nil

Equivalent to link(2).

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_symlink({path}, {new_path} [, {flags} [, {callback}]]) uv.fs_symlink()

Parameters:

- path: string
- new_path: string
- flags: table, integer, or nil
    - dir: boolean
    - junction: boolean
- callback: callable (async version) or nil (sync version)
    - err: nil or string
    - success: boolean or nil

Equivalent to symlink(2). If the flags parameter is omitted, then the 3rd parameter will be treated as the callback.

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_readlink({path} [, {callback}]) uv.fs_readlink()

Parameters:

- path: string
- callback: callable (async version) or nil (sync version)
    - err: nil or string
    - path: string or nil

Equivalent to readlink(2).

Returns (sync version): string or fail

Returns (async version): uv_fs_t userdata

uv.fs_realpath({path} [, {callback}]) uv.fs_realpath()

Parameters:

- path: string
- callback: callable (async version) or nil (sync version)
    - err: nil or string
    - path: string or nil

Equivalent to realpath(3).

Returns (sync version): string or fail

Returns (async version): uv_fs_t userdata

uv.fs_chown({path}, {uid}, {gid} [, {callback}]) uv.fs_chown()

Parameters:

- path: string
- uid: integer
- gid: integer
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - success: boolean or nil

Equivalent to chown(2).

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_fchown({fd}, {uid}, {gid} [, {callback}]) uv.fs_fchown()

Parameters:

- fd: integer
- uid: integer
- gid: integer
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - success: boolean or nil

Equivalent to fchown(2).

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_lchown({fd}, {uid}, {gid} [, {callback}]) uv.fs_lchown()

Parameters:

- fd: integer
- uid: integer
- gid: integer
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - success: boolean or nil

Equivalent to lchown(2).

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_copyfile({path}, {new_path} [, {flags} [, {callback}]]) uv.fs_copyfile()

Parameters:

- path: string

- new_path: string
- flags: table, integer, or nil
  - excl: boolean
  - ficlone: boolean
  - ficlone_force: boolean
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - success: boolean or nil

Copies a file from path to new_path. If the flags parameter is omitted, then the 3rd parameter will be treated as the callback.

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_opendir({path} [, {callback} [, {entries}]]) uv.fs_opendir()

Parameters:

- path: string
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - dir: luv_dir_t userdata or nil
- entries: integer or nil

Opens path as a directory stream. Returns a handle that the user can pass to uv.fs_readdir(). The entries parameter defines the maximum number of entries that should be returned by each call to uv.fs_readdir().

Returns (sync version): luv_dir_t userdata or fail

Returns (async version): uv_fs_t userdata

uv.fs_readdir({dir} [, {callback}]) uv.fs_readdir()

> method form dir:readdir([callback])

Parameters:

- dir: luv_dir_t userdata
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - entries: table or nil (see below)

Iterates over the directory stream luv_dir_t returned by a successful uv.fs_opendir() call. A table of data tables is returned where the number of entries n is equal to or less than the entries parameter used in the associated uv.fs_opendir() call.

Returns (sync version): table or fail

- [1, 2, 3, ..., n] : table
  - name : string
  - type : string

Returns (async version): uv_fs_t userdata

uv.fs_closedir({dir} [, {callback}]) uv.fs_closedir()

> method form dir:closedir([callback])

Parameters:

- dir: luv_dir_t userdata
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - success: boolean or nil

Closes a directory stream returned by a successful uv.fs_opendir() call.

Returns (sync version): boolean or fail

Returns (async version): uv_fs_t userdata

uv.fs_statfs({path} [, {callback}]) uv.fs_statfs()

Parameters:

- path: string
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - table or nil (see below)

Equivalent to statfs(2).

Returns table or nil

- type : integer
- bsize : integer
- blocks : integer
- bfree : integer
- bavail : integer
- files : integer
- ffree : integer
- frsize: integer or nil

THREAD POOL WORK SCHEDULING luv-thread-pool-work-scheduling

Libuv provides a threadpool which can be used to run user code and get notified in the loop thread. This threadpool is internally used to run all file system operations, as well as getaddrinfo and getnameinfo requests.

```
local function work_callback(a, b)
  return a + b
end

local function after_work_callback(c)
  print("The result is: " .. c)
end

local work = uv.new_work(work_callback, after_work_callback)

work:queue(1, 2)

-- output: "The result is: 3"
```

uv.new_work({work_callback}, {after_work_callback}) uv.new_work()

Parameters:

- work_callback: function or string
  - ...: threadargs passed to/from

uv.queue_work(work_ctx, ...)

- after_work_callback: function
  - ...: threadargs returned from work_callback

Creates and initializes a new luv_work_ctx_t (not uv_work_t).
work_callback is a Lua function or a string containing Lua code or bytecode dumped from a function. Returns the Lua userdata wrapping it.

Returns: luv_work_ctx_t userdata

uv.queue_work({work_ctx}, {...}) uv.queue_work()

> method form work_ctx:queue(...)

Parameters:

- work_ctx: luv_work_ctx_t userdata
- ...: threadargs

Queues a work request which will run work_callback in a new Lua state in a thread from the threadpool with any additional arguments from .... Values returned from work_callback are passed to after_work_callback, which is called in the main loop thread.

Returns: boolean or fail

DNS UTILITY FUNCTIONS luv-dns-utility-functions

uv.getaddrinfo({host}, {service} [, {hints} [, {callback}]]) uv.getaddrinfo()

Parameters:

- host: string or nil
- service: string or nil
- hints: table or nil
  - family: string or integer or nil
  - socktype: string or integer or nil
  - protocol: string or integer or nil
  - addrconfig: boolean or nil
  - v4mapped: boolean or nil
  - all: boolean or nil
  - numerichost: boolean or nil
  - passive: boolean or nil
  - numericserv: boolean or nil
  - canonname: boolean or nil
- callback: callable (async version) or nil (sync version)
  - err: nil or string

- addresses: table or nil (see below)

  Equivalent to getaddrinfo(3). Either node or service may
  be nil but not both.

  See luv-constants for supported address family input and
  output values.

  See luv-constants for supported socktype input and
  output values.

  When protocol is set to 0 or nil, it will be
  automatically chosen based on the socket's domain and type.
  When protocol is specified as a string, it will be looked up
  using the getprotobyname(3) function. Examples: "ip",
  "icmp", "tcp", "udp", etc.

  Returns (sync version): table or fail
- [1, 2, 3, ..., n] : table
  - addr : string
  - family : string
  - port : integer or nil
  - socktype : string
  - protocol : string
  - canonname : string or nil

  Returns (async version): uv_getaddrinfo_t userdata or fail

uv.getnameinfo({address} [, {callback}]) uv.getnameinfo()

  Parameters:
- address: table
  - ip: string or nil
  - port: integer or nil
  - family: string or integer or nil
- callback: callable (async version) or nil (sync
  version)
  - err: nil or string
  - host: string or nil
  - service: string or nil

  Equivalent to getnameinfo(3).

  See luv-constants for supported address family input values.

  Returns (sync version): string, string or fail

  Returns (async version): uv_getnameinfo_t userdata or fail

THREADING AND SYNCHRONIZATION UTILITIES luv-threading-and-synchronization-utilities

Libuv provides cross-platform implementations for multiple threading and
synchronization primitives. The API largely follows the pthreads API.

uv.new_thread([{options},] {entry}, {...}) uv.new_thread()

  Parameters:
- options: table or nil
  - stack_size: integer or nil
- entry: function or string

- ...: threadargs passed to entry

Creates and initializes a luv_thread_t (not uv_thread_t). Returns the Lua userdata wrapping it and asynchronously executes entry, which can be either a Lua function or a string containing Lua code or bytecode dumped from a function. Additional arguments ... are passed to the entry function and an optional options table may be provided. Currently accepted option fields are stack_size.

Returns: luv_thread_t userdata

Note: unsafe, please make sure that the thread's end of life is before Lua state is closed.

uv.thread_equal({thread}, {other_thread}) uv.thread_equal()

> method form thread:equal(other_thread)

Parameters:

- thread: luv_thread_t userdata
- other_thread: luv_thread_t userdata

Returns a boolean indicating whether two threads are the same. This function is equivalent to the __eq metamethod.

Returns: boolean

uv.thread_setaffinity()

uv.thread_setaffinity({thread}, {affinity} [, {get_old_affinity}])

> method form thread:setaffinity(affinity, [get_old_affinity])

Parameters:

- thread: luv_thread_t userdata
- affinity: table
  - [1, 2, 3, ..., n] : boolean
- get_old_affinity: boolean

Sets the specified thread's affinity setting.

affinity must be a table where each of the keys are a CPU number and the values are booleans that represent whether the thread should be eligible to run on that CPU. If the length of the affinity table is not greater than or equal to uv.cpumask_size(), any CPU numbers missing from the table will have their affinity set to false. If setting the affinity of more than uv.cpumask_size() CPUs is desired, affinity must be an array-like table with no gaps, since #affinity will be used as the cpumask_size if it is greater than uv.cpumask_size().

If get_old_affinity is true, the previous affinity settings for the thread will be returned. Otherwise, true is returned after a successful call.

Note: Thread affinity setting is not atomic on Windows. Unsupported on macOS.

Returns: table or boolean or fail

- [1, 2, 3, ..., n] : boolean

uv.thread_getaffinity({thread} [, {mask_size}]) uv.thread_getaffinity()

> method form thread:getaffinity([mask_size])

Parameters:

- thread: luv_thread_t userdata
- mask_size: integer

Gets the specified thread's affinity setting.

If mask_size is provided, it must be greater than or equal to uv.cpumask_size(). If the mask_size parameter is omitted, then the return of uv.cpumask_size() will be used. Returns an array-like table where each of the keys correspond to a CPU number and the values are booleans that represent whether the thread is eligible to run on that CPU.

Note: Thread affinity getting is not atomic on Windows. Unsupported on macOS.

Returns: table or fail

- [1, 2, 3, ..., n] : boolean

uv.thread_getcpu() uv.thread_getcpu()

Gets the CPU number on which the calling thread is running.

Note: The first CPU will be returned as the number 1, not 0. This allows for the number to correspond with the table keys used in uv.thread_getaffinity and uv.thread_setaffinity.

Returns: integer or fail

uv.thread_setpriority({thread}, {priority}) uv.thread.setpriority()

> method form thread:setpriority(priority)

Parameters:

- thread: luv_thread_t userdata
- priority: number

Sets the specified thread's scheduling priority setting. It requires elevated privilege to set specific priorities on some platforms. The priority can be set to the following constants:

- uv.constants.THREAD_PRIORITY_HIGHEST
- uv.constants.THREAD_PRIORITY_ABOVE_NORMAL
- uv.constants.THREAD_PRIORITY_NORMAL
- uv.constants.THREAD_PRIORITY_BELOW_NORMAL
- uv.constants.THREAD_PRIORITY_LOWEST

Returns: boolean or fail

uv.thread_getpriority({thread}) uv.thread.getpriority()

> method form thread:getpriority()

Parameters:

- thread: luv_thread_t userdata

Gets the thread's priority setting.

Retrieves the scheduling priority of the specified thread. The returned priority value is platform dependent.

For Linux, when schedule policy is SCHED_OTHER (default), priority is 0.

Returns: number or fail

uv.thread_self() uv.thread_self()

Returns the handle for the thread in which this is called.

Returns: luv_thread_t

uv.thread_join({thread}) uv.thread_join()

> method form thread:join()

Parameters:

- thread: luv_thread_t userdata

Waits for the thread to finish executing its entry function.

Returns: boolean or fail

uv.thread_detach({thread}) uv.thread_detach()

> method form thread:detach()

Parameters:

- thread: luv_thread_t userdata

Detaches a thread. Detached threads automatically release their resources upon termination, eliminating the need for the application to call uv.thread_join.

Returns: boolean or fail

uv.thread_setname({name}) uv.thread_setname()

Parameters:

- name: string

Sets the name of the current thread. Different platforms define different limits on the max number of characters a thread name can be: Linux, IBM i (16), macOS (64), Windows (32767), and NetBSD (32), etc. The name will be truncated if name is larger than the limit of the platform.

Returns: 0 or fail

uv.thread_getname({thread}) uv.thread_getname()

> method form thread:getname()

Parameters:

- thread: luv_thread_t userdata

Gets the name of the thread specified by thread.

Returns: string or fail

uv.sleep({msec}) uv.sleep()

Parameters:

- msec: integer

Pauses the thread in which this is called for a number of milliseconds.

Returns: Nothing.

uv.new_sem([{value}]) uv.new_sem()

Parameters:

- value: integer or nil

Creates a new semaphore with the specified initial value. A semaphore is safe to share across threads. It represents an unsigned integer value that can incremented and decremented atomically but any attempt to make it negative will "wait" until the value can be decremented by another thread incrementing it.

The initial value must be a non-negative integer.

Returns: luv_sem_t userdata or fail

Note: A semaphore must be shared between threads, any uv.sem_wait() on a single thread that blocks will deadlock.

uv.sem_post({sem}) uv.sem_post()

> method form sem:post()

Parameters:

- sem: luv_sem_t userdata

Increments (unlocks) a semaphore, if the semaphore's value consequently becomes greater than zero then another thread blocked in a sem_wait call will be woken and proceed to decrement the semaphore.

Returns: Nothing.

uv.sem_wait({sem}) uv.sem_wait()

> method form sem:wait()

Parameters:

- sem: luv_sem_t userdata

Decrements (locks) a semaphore, if the semaphore's value is greater than zero then the value is decremented and the call returns immediately. If the semaphore's value is zero then the call blocks until the semaphore's value rises above zero or the call is interrupted by a signal.

Returns: Nothing.

uv.sem_trywait({sem}) uv.sem_trywait()

> method form sem:trywait()

Parameters:

- sem: luv_sem_t userdata

The same as uv.sem_wait() but returns immediately if the semaphore is not available.

If the semaphore's value was decremented then true is returned, otherwise the semaphore has a value of zero and false is returned.

Returns: boolean

MISCELLANEOUS UTILITIES luv-miscellaneous-utilities

uv.exepath() uv.exepath()

Returns the executable path.

Returns: string or fail

uv.cwd() uv.cwd()

Returns the current working directory.

Returns: string or fail

uv.chdir({cwd}) uv.chdir()

Parameters:

- cwd: string

Sets the current working directory with the string cwd.

Returns: 0 or fail

uv.get_process_title() uv.get_process_title()

Returns the title of the current process.

Returns: string or fail

uv.set_process_title({title}) uv.set_process_title()

Parameters:

- title: string

Sets the title of the current process with the string title.

Returns: 0 or fail

uv.get_total_memory() uv.get_total_memory()

Returns the current total system memory in bytes.

Returns: number

uv.get_free_memory() uv.get_free_memory()

Returns the current free system memory in bytes.

Returns: number

uv.get_constrained_memory() uv.get_constrained_memory()

Gets the amount of memory available to the process in bytes based on limits imposed by the OS. If there is no such constraint, or the constraint is unknown, 0 is returned. Note that it is not unusual for this value to be less than or greater than the total system memory.

Returns: number

uv.get_available_memory() uv.get_available_memory()

Gets the amount of free memory that is still available to the process (in bytes). This differs from uv.get_free_memory() in that it takes into account any limits imposed by the OS. If there is no such constraint, or the constraint is unknown, the amount returned will be identical to uv.get_free_memory().

Returns: number

uv.resident_set_memory() uv.resident_set_memory()

Returns the resident set size (RSS) for the current process.

Returns: integer or fail

uv.getrusage() uv.getrusage()

Returns the resource usage.

Returns: table or fail

- utime : table (user CPU time used)
    - sec : integer
    - usec : integer
- stime : table (system CPU time used)
    - sec : integer
    - usec : integer
- maxrss : integer (maximum resident set size)
- ixrss : integer (integral shared memory size)
- idrss : integer (integral unshared data size)

- isrss : integer (integral unshared stack size)
- minflt : integer (page reclaims (soft page faults))
- majflt : integer (page faults (hard page faults))
- nswap : integer (swaps)
- inblock : integer (block input operations)
- oublock : integer (block output operations)
- msgsnd : integer (IPC messages sent)
- msgrcv : integer (IPC messages received)
- nsignals : integer (signals received)
- nvcsw : integer (voluntary context switches)
- nivcsw : integer (involuntary context switches)

uv.getrusage_thread() uv.getrusage_thread()

Gets the resource usage measures for the calling thread.

Note: Not supported on all platforms. May return ENOTSUP.

On macOS and Windows not all fields are set (the unsupported fields are filled with zeroes).

Returns: table or fail

- utime : table (user CPU time used)
  - sec : integer
  - usec : integer
- stime : table (system CPU time used)
  - sec : integer
  - usec : integer
- maxrss : integer (maximum resident set size)
- ixrss : integer (integral shared memory size)
- idrss : integer (integral unshared data size)
- isrss : integer (integral unshared stack size)
- minflt : integer (page reclaims (soft page faults))
- majflt : integer (page faults (hard page faults))
- nswap : integer (swaps)
- inblock : integer (block input operations)
- oublock : integer (block output operations)
- msgsnd : integer (IPC messages sent)

- msgrcv : integer (IPC messages received)
- nsignals : integer (signals received)
- nvcsw : integer (voluntary context switches)
- nivcsw : integer (involuntary context switches)

uv.available_parallelism() uv.available_parallelism()

Returns an estimate of the default amount of parallelism a program should use. Always returns a non-zero value.

On Linux, inspects the calling thread's CPU affinity mask to determine if it has been pinned to specific CPUs.

On Windows, the available parallelism may be underreported on systems with more than 64 logical CPUs.

On other platforms, reports the number of CPUs that the operating system considers to be online.

Returns: integer

uv.cpu_info() uv.cpu_info()

Returns information about the CPU(s) on the system as a table of tables for each CPU found.

Returns: table or fail

- [1, 2, 3, ..., n] : table
  - model : string
  - speed : number
  - times : table
    - user : number
    - nice : number
    - sys : number
    - idle : number
    - irq : number

uv.cpumask_size() uv.cpumask_size()

Returns the maximum size of the mask used for process/thread affinities, or ENOTSUP if affinities are not supported on the current platform.

Returns: integer or fail

uv.getpid() uv.getpid()

DEPRECATED: Please use uv.os_getpid() instead.

uv.getuid() uv.getuid()

Returns the user ID of the process.

Returns: integer

Note: This is not a libuv function and is not supported on Windows.

uv.getgid() uv.getgid()

Returns the group ID of the process.

Returns: integer

Note: This is not a libuv function and is not supported on Windows.

uv.setuid({id}) uv.setuid()

Parameters:

- id: integer

Sets the user ID of the process with the integer id.

Returns: Nothing.

Note: This is not a libuv function and is not supported on Windows.

uv.setgid({id}) uv.setgid()

Parameters:

- id: integer

Sets the group ID of the process with the integer id.

Returns: Nothing.

Note: This is not a libuv function and is not supported on Windows.

uv.hrtime() uv.hrtime()

Returns a current high-resolution time in nanoseconds as a number. This is relative to an arbitrary time in the past. It is not related to the time of day and therefore not subject to clock drift. The primary use is for measuring time between intervals.

Returns: integer

uv.clock_gettime({clock_id}) uv.clock_gettime()

Parameters:

- clock_id: string

Obtain the current system time from a high-resolution real-time or monotonic clock source. clock_id can be the string "monotonic" or "realtime".

The real-time clock counts from the UNIX epoch (1970-01-01) and is subject to time adjustments; it can jump back in time.

The monotonic clock counts from an arbitrary point in the past and never jumps back in time.

Returns: table or fail

- sec: integer
- nsec: integer

uv.uptime() uv.uptime()

Returns the current system uptime in seconds.

Returns: number or fail

uv.print_all_handles() uv.print_all_handles()

Prints all handles associated with the main loop to stderr. The format is [flags] handle-type handle-address . Flags are R for referenced, A for active and I for internal.

Returns: Nothing.

Note: This is not available on Windows.

WARNING: This function is meant for ad hoc debugging, there are no API/ABI stability guarantees.

uv.print_active_handles() uv.print_active_handles()

The same as uv.print_all_handles() except only active handles are printed.

Returns: Nothing.

Note: This is not available on Windows.

WARNING: This function is meant for ad hoc debugging, there are no API/ABI stability guarantees.

uv.guess_handle({fd}) uv.guess_handle()

Parameters:

- fd: integer

Used to detect what type of stream should be used with a given file descriptor fd. Usually this will be used during initialization to guess the type of the stdio streams.

Returns: string

uv.gettimeofday() uv.gettimeofday()

Cross-platform implementation of gettimeofday(2). Returns the seconds and microseconds of a unix time as a pair.

Returns: integer, integer or fail

uv.interface_addresses() uv.interface_addresses()

Returns address information about the network interfaces on the system in a table. Each table key is the name of the interface while each associated value is an array of address information where fields are ip, family, netmask, internal, and mac.

See luv-constants for supported address family output values.

Returns: table or fail

- [name(s)] : table
  - ip : string
  - family : string
  - netmask : string
  - internal : boolean
  - mac : string

uv.if_indextoname({ifindex}) uv.if_indextoname()

Parameters:

- ifindex: integer

IPv6-capable implementation of if_indextoname(3).

Returns: string or fail

uv.if_indextoiid({ifindex}) uv.if_indextoiid()

Parameters:

- ifindex: integer

Retrieves a network interface identifier suitable for use in an IPv6 scoped address. On Windows, returns the numeric ifindex as a string. On all other platforms, uv.if_indextoname() is used.

Returns: string or fail

uv.loadavg() uv.loadavg()

Returns the load average as a triad. Not supported on Windows.

Returns: number, number, number

uv.os_uname() uv.os_uname()

Returns system information.

Returns: table

- sysname : string
- release : string
- version : string
- machine : string

uv.os_gethostname() uv.os_gethostname()

Returns the hostname.

Returns: string

uv.os_getenv({name} [, {size}]) uv.os_getenv()

Parameters:

- name: string
- size: integer (default = LUAL_BUFFERSIZE)

Returns the environment variable specified by name as string. The internal buffer size can be set by defining size. If omitted, LUAL_BUFFERSIZE is used. If the environment variable exceeds the storage available in the internal buffer, ENOBUFS is returned. If no matching environment variable exists, ENOENT is returned.

Returns: string or fail

WARNING: This function is not thread safe.

uv.os_setenv({name}, {value}) uv.os_setenv()

Parameters:

- name: string

- value: string

Sets the environmental variable specified by name with the string value.

Returns: boolean or fail

WARNING: This function is not thread safe.

uv.os_unsetenv({name}) uv.os_unsetenv()

Parameters:

- name: string

Unsets the environmental variable specified by name.

Returns: boolean or fail

WARNING: This function is not thread safe.

uv.os_environ() uv.os_environ()

Returns all environmental variables as a dynamic table of names associated with their corresponding values.

Returns: table

WARNING: This function is not thread safe.

uv.os_homedir() uv.os_homedir()

Returns: string or fail

WARNING: This function is not thread safe.

uv.os_tmpdir() uv.os_tmpdir()

Returns: string or fail

WARNING: This function is not thread safe.

uv.os_get_passwd() uv.os_get_passwd()

Returns password file information.

Returns: table or fail

- username : string

- uid : integer? (nil on Windows)

- gid : integer? (nil on Windows)

- shell : string?(nil` on Windows)

- homedir : string

uv.os_getpid() uv.os_getpid()

Returns the current process ID.

Returns: number

uv.os_getppid() uv.os_getppid()

Returns the parent process ID.

Returns: number

uv.os_getpriority({pid}) uv.os_getpriority()

Parameters:

- pid: integer

Returns the scheduling priority of the process specified by pid.

Returns: number or fail

uv.os_setpriority({pid}, {priority}) uv.os_setpriority()

Parameters:

- pid: integer

- priority: integer

Sets the scheduling priority of the process specified by pid. The priority range is between -20 (high priority) and 19 (low priority).

Returns: boolean or fail

uv.random({len}, {flags} [, {callback}]) uv.random()

Parameters:

- len: integer

- flags: nil (see below)
- callback: callable (async version) or nil (sync version)
  - err: nil or string
  - bytes: string or nil

Fills a string of length len with cryptographically strong random bytes acquired from the system CSPRNG. flags is reserved for future extension and must currently be nil or 0 or {}.

Short reads are not possible. When less than len random bytes are available, a non-zero error value is returned or passed to the callback. If the callback is omitted, this function is completed synchronously.

The synchronous version may block indefinitely when not enough entropy is available. The asynchronous version may not ever finish when the system is low on entropy.

Returns (sync version): string or fail

Returns (async version): 0 or fail

uv.translate_sys_error({errcode}) uv.translate_sys_error()

Parameters:
- errcode: integer

Returns the libuv error message and error name (both in string form, see err and name in luv-error-handling) equivalent to the given platform dependent error code: POSIX error codes on Unix (the ones stored in errno), and Win32 error codes on Windows (those returned by GetLastError() or WSAGetLastError()).

Returns: string, string or nil

METRICS OPERATIONS luv-metrics-operations

uv.metrics_idle_time() uv.metrics_idle_time()

Retrieve the amount of time the event loop has been idle in the kernel's event provider (e.g. epoll_wait). The call is thread safe.

The return value is the accumulated time spent idle in the kernel's event provider starting from when the uv_loop_t was configured to collect the idle time.

Note: The event loop will not begin accumulating the event provider's idle time until calling loop_configure with "metrics_idle_time".

Returns: number

uv.metrics_info() uv.metrics_info()

Get the metrics table from current set of event loop metrics. It is recommended to retrieve these metrics in a prepare callback (see uv.new_prepare(), uv.prepare_start()) in order to make sure there are no inconsistencies with the metrics counters.

Returns: table

- loop_count : integer
- events : integer
- events_waiting : integer

STRING MANIPULATION FUNCTIONS luv-string-manipulation

These string utilities are needed internally for dealing with Windows, and are exported to allow clients to work uniformly with this data when the libuv API is not complete.

Notes:
1. New in luv version 1.49.0.
2. See the WTF-8 spec (https://simonsapin.github.io/wtf-8/) for information

about WTF-8.

3. Luv uses Lua-style strings, which means that all inputs and return values (UTF-8 or UTF-16 strings) do not include a NUL terminator.

uv.utf16_length_as_wtf8({utf16}) uv.utf16_length_as_wtf8()

Get the length (in bytes) of a UTF-16 (or UCS-2) string utf16 value after converting it to WTF-8. The endianness of the UTF-16 (or UCS-2) string is assumed to be the same as the native endianness of the platform.

Parameters:

- utf16: string

Returns: integer

uv.utf16_to_wtf8({utf16}) uv.utf16_to_wtf8()

Convert UTF-16 (or UCS-2) string utf16 to UTF-8 string. The endianness of the UTF-16 (or UCS-2) string is assumed to be the same as the native endianness of the platform.

Parameters:

- utf16: string

Returns: string

uv.wtf8_length_as_utf16({wtf16}) uv.wtf8_length_as_utf16()

Get the length (in UTF-16 code units) of a WTF-8 wtf8 value after converting it to UTF-16 (or UCS-2).

Note: The number of bytes needed for a UTF-16 (or UCS-2) string is <number of code units> * 2.

Parameters:

- wtf8: string

Returns: integer

uv.wtf8_to_utf16({wtf16}) uv.wtf8_to_utf16()

Convert WTF-8 string in wtf8 to UTF-16 (or UCS-2) string. The endianness of the UTF-16 (or UCS-2) string is assumed to be the same as the native endianness of the platform.

Parameters:

- wtf8: string

Returns: string

CREDITS luv-credits

This document is a reformatted version of the LUV documentation, up-to-date with commit dcd1a1c (23 Aug 2023) of the luv repository https://github.com/luvit/luv/commit/dcd1a1cad5b05634a7691402d6ca2f214fb4ae76.

Based on https://github.com/nanotee/luv-vimdocs with kind permission.

Key mapping, abbreviations and user-defined commands.

This subject is introduced in sections 05.3, 24.7 and 40.1 of the user manual.

1. Key mapping

key-mapping mapping

Key mapping is used to change the meaning of typed keys. The most common use is to define a sequence of commands for a function key. Example:

```
:map <F2> a<C-R>=strftime("%c")<CR><Esc>
```

This appends the current date and time after the cursor (in <> notation <>).

1.1 MAP COMMANDS

:map-commands

There are commands to enter new mappings, remove mappings and list mappings. See map-overview for the various forms of "map" and their relationships with modes.

{lhs} means left-hand-side {lhs}
{rhs} means right-hand-side {rhs}

:map {lhs} {rhs} mapmode-nvo :map

:nm[ap] {lhs} {rhs} mapmode-n :nm :nmap
:vm[ap] {lhs} {rhs} mapmode-v :vm :vmap
:xm[ap] {lhs} {rhs} mapmode-x :xm :xmap
:smap {lhs} {rhs} mapmode-s :smap

:om[ap] {lhs} {rhs} mapmode-o :om :omap
:map! {lhs} {rhs} mapmode-ic :map!

:im[ap] {lhs} {rhs} mapmode-i :im :imap
:lm[ap] {lhs} {rhs} mapmode-l :lm :lma :lmap
:cm[ap] {lhs} {rhs} mapmode-c :cm :cmap
:tma[p] {lhs} {rhs} mapmode-t :tma :tmap

Map the key sequence {lhs} to {rhs} for the modes where the map command applies. The result, including {rhs}, is then further scanned for mappings. This allows for nested and recursive use of mappings.
Note: Trailing spaces are included in the {rhs}, because space is a valid Normal mode command.
See map-trailing-white.

:nore :norem
:no[remap] {lhs} {rhs} mapmode-nvo :no :noremap :nor
:nn[oremap] {lhs} {rhs} mapmode-n :nn :nnoremap
:vn[oremap] {lhs} {rhs} mapmode-v :vn :vnoremap
:xn[oremap] {lhs} {rhs} mapmode-x :xn :xnoremap
:snor[emap] {lhs} {rhs} mapmode-s :snor :snore :snoremap
:ono[remap] {lhs} {rhs} mapmode-o :ono :onoremap
:no[remap]! {lhs} {rhs} mapmode-ic :no! :noremap!
:ino[remap] {lhs} {rhs} mapmode-i :ino :inor :inoremap
:ln[oremap] {lhs} {rhs} mapmode-l :ln :lnoremap
:cno[remap] {lhs} {rhs} mapmode-c :cno :cnor :cnoremap
:tno[remap] {lhs} {rhs} mapmode-t :tno :tnoremap

Map the key sequence {lhs} to {rhs} for the modes where the map command applies. Disallow mapping of {rhs}, to avoid nested and recursive mappings. Often used to redefine a command.
Note: Keys in {rhs} also won't trigger abbreviation, with the exception of i_CTRL-] and c_CTRL-].
Note: When <Plug> appears in the {rhs} this part is always applied even if remapping is disallowed.

:unm[ap] {lhs} mapmode-nvo :unm :unmap
:nun[map] {lhs} mapmode-n :nun :nunmap
:vu[nmap] {lhs} mapmode-v :vu :vunmap
:xu[nmap] {lhs} mapmode-x :xu :xunmap
:sunm[ap] {lhs} mapmode-s :sunm :sunmap
:ou[nmap] {lhs} mapmode-o :ou :ounmap
:unm[ap]! {lhs} mapmode-ic :unm! :unmap!
:iu[nmap] {lhs} mapmode-i :iu :iunmap
:lu[nmap] {lhs} mapmode-l :lu :lunmap
:cu[nmap] {lhs} mapmode-c :cu :cun :cunmap
:tunma[p] {lhs} mapmode-t :tunma :tunmap

Remove the mapping of {lhs} for the modes where the map command applies. The mapping may remain defined for other modes where it applies.
If no mapping with {lhs} exists, it also works when {lhs} matches the {rhs} of a mapping (see vim.keymap.del() to avoid that). This is for when an abbreviation applied.
Note: Trailing spaces are included in the {lhs}.
See map-trailing-white.

:mapc[lear] mapmode-nvo :mapc :mapclear
:nmapc[lear] mapmode-n :nmapc :nmapclear
:vmapc[lear] mapmode-v :vmapc :vmapclear
:xmapc[lear] mapmode-x :xmapc :xmapclear
:smapc[lear] mapmode-s :smapc :smapclear

:omapc[lear]		mapmode-o		:omapc	:omapclear
:mapc[lear]!		mapmode-ic		:mapc!	:mapclear!
:imapc[lear]		mapmode-i		:imapc	:imapclear
:lmapc[lear]		mapmode-l		:lmapc	:lmapclear
:cmapc[lear]		mapmode-c		:cmapc	:cmapclear
:tmapc[lear]		mapmode-t		:tmapc	:tmapclear

Remove ALL mappings for the modes where the map
command applies.
Use the <buffer> argument to remove buffer-local
mappings :map-<buffer>
Warning: This also removes the default-mappings.

:map		mapmode-nvo
:nm[ap]		mapmode-n
:vm[ap]		mapmode-v
:xm[ap]		mapmode-x
:sm[ap]		mapmode-s
:om[ap]		mapmode-o
:map!		mapmode-ic
:im[ap]		mapmode-i
:lm[ap]		mapmode-l
:cm[ap]		mapmode-c
:tma[p]		mapmode-t

List all key mappings for the modes where the map
command applies. Note that ":map" and ":map!" are
used most often, because they include the other modes.

:map {lhs}		mapmode-nvo		:map_l

:nm[ap] {lhs}		mapmode-n		:nmap_l

:vm[ap] {lhs}		mapmode-v		:vmap_l

:xm[ap] {lhs}		mapmode-x		:xmap_l

:sm[ap] {lhs}		mapmode-s		:smap_l

:om[ap] {lhs}		mapmode-o		:omap_l

:map! {lhs}		mapmode-ic		:map_l!

:im[ap] {lhs}		mapmode-i		:imap_l

:lm[ap] {lhs}		mapmode-l		:lmap_l

:cm[ap] {lhs}		mapmode-c		:cmap_l

:tma[p] {lhs}		mapmode-t		:tmap_l

List the key mappings for the key sequences starting
with {lhs} in the modes where the map command applies.

These commands are used to map a key or key sequence to a string of
characters. You can use this to put command sequences under function keys,
translate one key into another, etc. See :mkexrc for how to save and
restore the current mappings.

map-ambiguous

When two mappings start with the same sequence of characters, they are
ambiguous. Example:

```
:imap aa foo
:imap aaa bar
```

When Vim has read "aa", it will need to get another character to be able to
decide if "aa" or "aaa" should be mapped. This means that after typing "aa"
that mapping won't get expanded yet, Vim is waiting for another character.
If you type a space, then "foo" will get inserted, plus the space. If you
type "a", then "bar" will get inserted.

Trailing white space

map-trailing-white

This unmap command does NOT work:

```
:map @@ foo
:unmap @@ | print
```

Because it tries to unmap "@@ ", including the white space before the command
separator "|". Other examples with trailing white space:

```
unmap @@
unmap @@     " comment
```

An error will be issued, which is very hard to identify, because the ending
whitespace character in unmap @@ is not visible.

A generic solution is to put the command separator "|" right after the mapped
keys. After that white space and a comment may follow:

```
unmap @@|    " comment
```

1.2 SPECIAL ARGUMENTS

:map-arguments

"<buffer>", "<nowait>", "<silent>", "<script>", "<expr>" and
"<unique>" can be used in any order. They must appear right after the
command, before any other arguments.

:map-local :map-<buffer> :map-buffer
E224 E225

If the first argument to one of these commands is "<buffer>" the mapping will
be effective in the current buffer only. Example:

```
:map <buffer>  ,w  /[.,;]<CR>
```

Then you can map ",w" to something else in another buffer:

```
:map <buffer>  ,w  /[#&!]<CR>
```

The local buffer mappings are used before the global ones. See <nowait> below
to make a short local mapping take effect when a longer global one exists.
The "<buffer>" argument can also be used to clear mappings:

```
:unmap <buffer> ,w
:mapclear <buffer>
```

Local mappings are also cleared when a buffer is deleted, but not when it is
unloaded. Just like local option values.
Also see map-precedence.

:map-<nowait> :map-nowait

When defining a buffer-local mapping for "," there may be a global mapping
that starts with ",". Then you need to type another character for Vim to know
whether to use the "," mapping or the longer one. To avoid this add the
<nowait> argument. Then the mapping will be used when it matches, Vim does
not wait for more characters to be typed. However, if the characters were
already typed they are used.
Note that this works when the <nowait> mapping fully matches and is found
before any partial matches. This works when:

- There is only one matching buffer-local mapping, since these are always
  found before global mappings.
- There is another buffer-local mapping that partly matches, but it is
  defined earlier (last defined mapping is found first).

:map-<silent> :map-silent

To define a mapping which will not be echoed on the command line, add
"<silent>" as the first argument. Example:

```
:map <silent> ,h /Header<CR>
```

The search string will not be echoed when using this mapping. Messages from
the executed command are still given though. To shut them up too, add a
":silent" in the executed command:

```
:map <silent> ,h :exe ":silent normal /Header\r"<CR>
```

Note that the effect of a command might also be silenced, e.g., when the
mapping selects another entry for command line completion it won't be
displayed.
Prompts will still be given, e.g., for inputdialog().
Using "<silent>" for an abbreviation is possible, but will cause redrawing of
the command line to fail.

:map-<script> :map-script

If the first argument to one of these commands is "<script>" and it is used to
define a new mapping or abbreviation, the mapping will only remap characters
in the {rhs} using mappings that were defined local to a script, starting with
"<SID>". This can be used to avoid that mappings from outside a script
interfere (e.g., when CTRL-V is remapped in mswin.vim), but do use other
mappings defined in the script.
Note: ":map <script>" and ":noremap <script>" do the same thing. The
"<script>" overrules the command name. Using ":noremap <script>" is
preferred, because it's clearer that remapping is (mostly) disabled.

:map-<unique> :map-unique E226 E227

If the first argument to one of these commands is "<unique>" and it is used to
define a new mapping or abbreviation, the command will fail if the mapping or
abbreviation already exists. Example:

```
:map <unique> ,w  /[#&!]<CR>
```

When defining a local mapping, there will also be a check if a global map
already exists which is equal.
Example of what will fail:

```
:map ,w  /[#&!]<CR>
:map <buffer> <unique> ,w  /[.,;]<CR>
```

If you want to map a key and then have it do what it was originally mapped to,
have a look at maparg().

:map-<expr> :map-expression

If the first argument to one of these commands is "<expr>" and it is used to
define a new mapping or abbreviation, the argument is an expression. The
expression is evaluated to obtain the {rhs} that is used. Example:

```
:inoremap <expr> . <SID>InsertDot()
```

The result of the s:InsertDot() function will be inserted. It could check the
text before the cursor and start omni completion when some condition is met.
Using a script-local function is preferred, to avoid polluting the global
namespace. Use <SID> in the RHS so that the script that the mapping was
defined in can be found.

For abbreviations v:char is set to the character that was typed to trigger
the abbreviation. You can use this to decide how to expand the {lhs}. You
should not either insert or change the v:char.

In case you want the mapping to not do anything, you can have the expression
evaluate to an empty string. If something changed that requires Vim to
go through the main loop (e.g. to update the display), return "\<Ignore>".
This is similar to "nothing" but makes Vim return from the loop that waits for
input.

Keep in mind that the expression may be evaluated when looking for
typeahead, before the previous command has been executed. For example:

```
func StoreColumn()
  let g:column = col('.')
  return 'x'
endfunc
nnoremap <expr> x StoreColumn()
nmap ! f!x
```

You will notice that g:column has the value from before executing "f!",
because "x" is evaluated before "f!" is executed.
This can be solved by inserting <Ignore> before the character that is
expression-mapped:

```
nmap ! f!<Ignore>x
```

Be very careful about side effects! The expression is evaluated while
obtaining characters, you may very well make the command dysfunctional.
Therefore the following is blocked for <expr> mappings:

- Changing the buffer text textlock.
- Editing another buffer.
- The :normal command.
- Moving the cursor is allowed, but it is restored afterwards.
- If the cmdline is changed, the old text and cursor position are restored.

  If you want the mapping to do any of these let the returned characters do
  that, or use a <Cmd> mapping instead.

You can use getchar(), it consumes typeahead if there is any. E.g., if you
have these mappings:

```
inoremap <expr> <C-L> nr2char(getchar())
inoremap <expr> <C-L>x "foo"
```

If you now type CTRL-L nothing happens yet, Vim needs the next character to
decide what mapping to use. If you type 'x' the second mapping is used and
"foo" is inserted. If you type any other key the first mapping is used,
getchar() gets the typed key and returns it.

Here is an example that inserts a list number that increases:

```
let counter = 0
inoremap <expr> <C-L> ListItem()
inoremap <expr> <C-R> ListReset()

func ListItem()
  let g:counter += 1
  return g:counter .. '. '
endfunc

func ListReset()
  let g:counter = 0
  return ''
endfunc
```

CTRL-L inserts the next number, CTRL-R resets the count. CTRL-R returns an empty string, so that nothing is inserted.

Note that using 0x80 as a single byte before other text does not work, it will be seen as a special key.

<Cmd> :map-cmd

The <Cmd> pseudokey begins a "command mapping", which executes the command directly without changing modes. Where you might use ":…<CR>" in the {rhs} of a mapping, you can instead use "<Cmd>…<CR>". Example:

```
noremap x <Cmd>echo mode(1)<CR>
```

This is more flexible than :<C-U> in Visual and Operator-pending mode, or <C-O>: in Insert mode, because the commands are executed directly in the current mode, instead of always going to Normal mode. Visual mode is preserved, so tricks with gv are not needed. Commands can be invoked directly in Command-line mode (which would otherwise require timer hacks). Example of using <Cmd> halfway Insert mode:

```
nnoremap <F3> aText <Cmd>echo mode(1)<CR> Added<Esc>
```

Unlike <expr> mappings, there are no special restrictions on the <Cmd> command: it is executed as if an (unrestricted) autocommand was invoked or an async event was processed.

Note:

- Because <Cmd> avoids mode-changes (unlike ":") it does not trigger CmdlineEnter and CmdlineLeave events. This helps performance.
- For the same reason, keycodes like <C-R><C-W> are interpreted as plain, unmapped keys.
- The command is not echo'd, no need for <silent>.
- The {rhs} is not subject to abbreviations nor to other mappings, even if the mapping is recursive.
- In Visual mode you can use line('v') and col('v') to get one end of the Visual area, the cursor is at the other end.

E1255 E1136

<Cmd> commands must terminate, that is, they must be followed by <CR> in the {rhs} of the mapping definition. Command-line mode is never entered. To use a literal "<" in the {rhs}, use <lt>.

1.3 MAPPING AND MODES :map-modes

mapmode-nvo mapmode-n mapmode-v mapmode-o mapmode-t

There are seven sets of mappings

- For Normal mode: When typing commands.
- For Visual mode: When typing commands while the Visual area is highlighted.
- For Select mode: like Visual mode but typing text replaces the selection.
- For Operator-pending mode: When an operator is pending (after "d", "y", "c", etc.). See below: omap-info.
- For Insert mode. These are also used in Replace mode.
- For Command-line mode: When entering a ":" or "/" command.

- For Terminal mode: When typing in a :terminal buffer.

Special case: While typing a count for a command in Normal mode, mapping zero is disabled. This makes it possible to map zero without making it impossible to type a count with a zero.

map-overview map-modes

Overview of which map command works in which mode. More details below.

```
COMMANDS                    MODES
:map   :noremap  :unmap     Normal, Visual, Select, Operator-pending
:nmap  :nnoremap :nunmap    Normal
:vmap  :vnoremap :vunmap    Visual and Select
:smap  :snoremap :sunmap    Select
:xmap  :xnoremap :xunmap    Visual
:omap  :onoremap :ounmap    Operator-pending
:map!  :noremap! :unmap!    Insert and Command-line
:imap  :inoremap :iunmap    Insert
:lmap  :lnoremap :lunmap    Insert, Command-line, Lang-Arg
:cmap  :cnoremap :cunmap    Command-line
:tmap  :tnoremap :tunmap    Terminal
```

Same information in a table:

map-table

```
Mode  | Norm | Ins | Cmd | Vis | Sel | Opr | Term | Lang |
Command        +------+-----+-----+-----+-----+-----+------+------+
[nore]map      | yes  |  -  |  -  | yes | yes | yes |  -   |  -   |
n[nore]map     | yes  |  -  |  -  |  -  |  -  |  -  |  -   |  -   |
[nore]map!     |  -   | yes | yes |  -  |  -  |  -  |  -   |  -   |
i[nore]map     |  -   | yes |  -  |  -  |  -  |  -  |  -   |  -   |
c[nore]map     |  -   |  -  | yes |  -  |  -  |  -  |  -   |  -   |
v[nore]map     |  -   |  -  |  -  | yes | yes |  -  |  -   |  -   |
x[nore]map     |  -   |  -  |  -  | yes |  -  |  -  |  -   |  -   |
s[nore]map     |  -   |  -  |  -  |  -  | yes |  -  |  -   |  -   |
o[nore]map     |  -   |  -  |  -  |  -  |  -  | yes |  -   |  -   |
t[nore]map     |  -   |  -  |  -  |  -  |  -  |  -  | yes  |  -   |
l[nore]map     |  -   | yes | yes |  -  |  -  |  -  |  -   | yes  |
```

COMMANDS	MODES		
Normal Visual+Select Operator-pending			
:map :noremap :unmap :mapclear	yes	yes	yes
:nmap :nnoremap :nunmap :nmapclear	yes	-	-
:vmap :vnoremap :vunmap :vmapclear	-	yes	-
:omap :onoremap :ounmap :omapclear	-	-	yes

:nunmap can also be used outside of a monastery.

mapmode-x mapmode-s

Some commands work both in Visual and Select mode, some in only one. Note that quite often "Visual" is mentioned where both Visual and Select mode apply. Select-mode-mapping

NOTE: Mapping a printable character in Select mode may confuse the user. It's better to explicitly use :xmap and :smap for printable characters. Or use :sunmap after defining the mapping.

COMMANDS	MODES	
Visual Select		
:vmap :vnoremap :vunmap :vmapclear	yes	yes
:xmap :xnoremap :xunmap :xmapclear	yes	-
:smap :snoremap :sunmap :smapclear	-	yes

mapmode-ic mapmode-i mapmode-c mapmode-l

Some commands work both in Insert mode and Command-line mode, some not:

COMMANDS	MODES		
Insert Command-line Lang-Arg			
:map! :noremap! :unmap! :mapclear!	yes	yes	-
:imap :inoremap :iunmap :imapclear	yes	-	-
:cmap :cnoremap :cunmap :cmapclear	-	yes	-
:lmap :lnoremap :lunmap :lmapclear	yes*	yes*	yes*

* If 'iminsert' is 1, see language-mapping below.

The original Vi did not have separate mappings for Normal/Visual/Operator-pending mode and for Insert/Command-line mode. Therefore the ":map" and ":map!" commands enter and display mappings for several modes. In Vim you can use the ":nmap", ":vmap", ":omap", ":cmap" and ":imap" commands to enter mappings for each mode separately.

omap-info

Operator-pending mappings can be used to define a movement command that can be used with any operator. Simple example:

```
:omap { w
```

makes "y{" work like "yw" and "d{" like "dw".

To ignore the starting cursor position and select different text, you can have the omap start Visual mode to select the text to be operated upon. Example that operates on a function name in the current line:

```
onoremap <silent> F :<C-U>normal! 0f(hviw<CR>
```

The CTRL-U (<C-U>) is used to remove the range that Vim may insert. The Normal mode commands find the first '(' character and select the first word (usually is the function name) before it. The operator does not change the Visual marks '< '>. gv

To enter a mapping for Normal and Visual mode, but not Operator-pending mode, first define it for all three modes, then unmap it for Operator-pending mode:

```
:map    xx something-difficult
:ounmap xx
```

Likewise for a mapping for Visual and Operator-pending mode or Normal and Operator-pending mode.

language-mapping

":lmap" defines a mapping that applies to:

- Insert mode
- Command-line mode
- when entering a search pattern
- the argument of the commands that accept a text character, such as "r" and "f"
- for the input() line

Generally: Whenever a character is to be typed that is part of the text in the buffer, not a Vim command character. "Lang-Arg" isn't really another mode, it's just used here for this situation.

The simplest way to load a set of related language mappings is by using the 'keymap' option. See 45.5.

In Insert mode and in Command-line mode the mappings can be disabled with the CTRL-^ command i_CTRL-^ c_CTRL-^. These commands change the value of the 'iminsert' option. When starting to enter a normal command line (not a search pattern) the mappings are disabled until a CTRL-^ is typed. The state last used is remembered for Insert mode and Search patterns separately. The state for Insert mode is also used when typing a character as an argument to command like "f" or "t".

Language mappings will never be applied to already mapped characters. They are only used for typed characters. This assumes that the language mapping was already done when typing the mapping. Correspondingly, language mappings are applied when recording macros, rather than when applying them.

1.4 LISTING MAPPINGS

map-listing

When listing mappings the characters in the first two columns are:

CHAR	MODE
<Space>	Normal, Visual, Select and Operator-pending
n	Normal
v	Visual and Select
s	Select
x	Visual
o	Operator-pending
!	Insert and Command-line
i	Insert
l	":lmap" mappings for Insert, Command-line and Lang-Arg
c	Command-line
t	Terminal-Job

Just before the {rhs} a special character can appear:

```
        *       indicates that it is not remappable
        &       indicates that only script-local mappings are remappable
        @       indicates a buffer-local mapping
```

Everything from the first non-blank after {lhs} up to the end of the line (or '|') is considered to be part of {rhs}. This allows the {rhs} to end with a space.

Note: When using mappings for Visual mode, you can use the "'<" mark, which is the start of the last selected Visual area in the current buffer '<.

The :filter command can be used to select what mappings to list. The pattern is matched against the {lhs} and {rhs} in the raw form. If a description was added using nvim_set_keymap() or nvim_buf_set_keymap() then the pattern is also matched against it.

:map-verbose

When 'verbose' is non-zero, listing a key map will also display where it was last defined. Example:

```
:verbose map <C-W>*
n  <C-W>*      * <C-W><C-S>*
        Last set from ~/.config/nvim/init.vim
```

See :verbose-cmd for more information.

1.5 MAPPING SPECIAL KEYS :map-special-keys

To map a function key, use the internal code for it. To enter such a mapping type CTRL-K and then hit the function key, or use the form "<F2>", "<F10>", "<Up>", "<S-Down>", "<S-F7>", etc. (see table of keys key-notation, all keys from <Up> can be used).

1.6 SPECIAL CHARACTERS :map-special-chars

map_backslash map-backslash
Note that only CTRL-V is mentioned here as a special character for mappings and abbreviations. When 'cpoptions' does not contain 'B', a backslash can also be used like CTRL-V. The <> notation can be fully used then <>. But you cannot use "<C-V>" like CTRL-V to escape the special meaning of what follows.

To map a backslash, or use a backslash literally in the {rhs}, the special sequence "<Bslash>" can be used. This avoids the need to double backslashes when using nested mappings.

map_CTRL-C map-CTRL-C
Using CTRL-C in the {lhs} is possible, but it will only work when Vim is waiting for a key, not when Vim is busy with something. When Vim is busy CTRL-C interrupts/breaks the command.
When using the GUI version on MS-Windows CTRL-C can be mapped to allow a Copy command to the clipboard. Use CTRL-Break to interrupt Vim.

map_space_in_lhs map-space_in_lhs
To include a space in {lhs} precede it with a CTRL-V (type two CTRL-Vs for each space).

map_space_in_rhs map-space_in_rhs
If you want a {rhs} that starts with a space, use "<Space>". To be fully Vi compatible (but unreadable) don't use the <> notation, precede {rhs} with a single CTRL-V (you have to type CTRL-V two times).

map_empty_rhs map-empty-rhs
You can create an empty {rhs} by typing nothing after a single CTRL-V (you have to type CTRL-V two times). Unfortunately, you cannot do this in a vimrc file.

<Nop>
An easier way to get a mapping that doesn't produce anything, is to use "<Nop>" for the {rhs}. For example, to disable function key 8:

```
:map  <F8>  <Nop>
:map! <F8>  <Nop>
```

map-multibyte

It is possible to map multibyte characters, but only the whole character. You cannot map the first byte only. This was done to prevent problems in this scenario:

```
:set encoding=latin1
:imap <M-C> foo
:set encoding=utf-8
```

The mapping for <M-C> is defined with the latin1 encoding, resulting in a 0xc3 byte. If you type the character á (0xe1 <M-a>) in UTF-8 encoding this is the two bytes 0xc3 0xa1. You don't want the 0xc3 byte to be mapped then or otherwise it would be impossible to type the á character.

<Leader> mapleader
To define a mapping which uses the "g:mapleader" variable, the special string "<Leader>" can be used. It is replaced with the string value of "g:mapleader". If "g:mapleader" is not set or empty, a backslash is used instead. Example:

```
map <Leader>A  oanother line<Esc>
```

Works like:

```
map \A  oanother line<Esc>
```

But after:

```
let mapleader = ","
```

It works like:

```
map ,A  oanother line<Esc>
```

Note that the value of "g:mapleader" is used at the moment the mapping is
defined. Changing "g:mapleader" after that has no effect for already defined
mappings.

<LocalLeader> maplocalleader

<LocalLeader> is just like <Leader>, except that it uses "maplocalleader"
instead of "mapleader". <LocalLeader> is to be used for mappings which are
local to a buffer. Example:

```
:map <buffer> <LocalLeader>A  oanother line<Esc>
```

In a global plugin <Leader> should be used and in a filetype plugin
<LocalLeader>. "mapleader" and "maplocalleader" can be equal. Although, if
you make them different, there is a smaller chance of mappings from global
plugins to clash with mappings for filetype plugins. For example, you could
keep "mapleader" at the default backslash, and set "maplocalleader" to an
underscore.

map-<SID>

In a script the special key name "<SID>" can be used to define a mapping
that's local to the script. See <SID> for details.

<Plug>

The special key name "<Plug>" can be used for an internal mapping, which is
not to be matched with any key sequence. This is useful in plugins
using-<Plug>.

<MouseMove>

The special key name "<MouseMove>" can be used to handle mouse movement. It
needs to be enabled with 'mousemoveevent'.
The getmousepos() function can be used to obtain the mouse position.

<Char> <Char->

To map a character by its decimal, octal or hexadecimal number the <Char>
construct can be used:

```
        <Char-123>      character 123
        <Char-033>      character 27
        <Char-0x7f>     character 127
        <S-Char-114>    character 114 ('r') shifted ('R')
```

This is useful to specify a (multibyte) character in a 'keymap' file.
Upper and lowercase differences are ignored.

map-comments

It is not possible to put a comment after these commands, because the "
character is considered to be part of the {lhs} or {rhs}. However, one can
use |", since this starts a new, empty command with a comment.

map_bar map-bar

Since the '|' character is used to separate a map command from the next
command, you will have to do something special to include a '|' in {rhs}.
There are three methods:

use	works when	example
<Bar>	always	:map _l :!ls <Bar> more^M
|	'b' is not in 'cpoptions'	:map _l :!ls | more^M
^V|	always	:map _l :!ls ^V| more^M

(here ^V stands for CTRL-V; to get one CTRL-V you have to type it twice; you
cannot use the <> notation "<C-V>" here).

All three work when you use the default setting for 'cpoptions'.

When 'b' is present in 'cpoptions', "\|" will be recognized as a mapping
ending in a '\' and then another command. This is Vi compatible, but
illogical when compared to other commands.

map_return map-return

When you have a mapping that contains an Ex command, you need to put a line
terminator after it to have it executed. The use of <CR> is recommended for
this (see <>). Example:

```
:map  _ls  :!ls -l %:S<CR>:echo "the end"<CR>
```

To avoid mapping of the characters you type in insert or Command-line mode,
type a CTRL-V first.

map-error

Note that when an error is encountered (that causes an error message or might
cause a beep) the rest of the mapping is not executed. This is Vi-compatible.

Note that the second character (argument) of the commands @zZtTfF[]rm'`"v
and CTRL-X is not mapped. This was done to be able to use all the named

registers and marks, even when the command with the same name has been
mapped.

1.7 WHAT KEYS TO MAP

map-which-keys

If you are going to map something, you will need to choose which key(s) to use
for the {lhs}. You will have to avoid keys that are used for Vim commands,
otherwise you would not be able to use those commands anymore. Here are a few
suggestions:

- Function keys <F2>, <F3>, etc.. Also the shifted function keys <S-F1>,
  <S-F2>, etc. Note that <F1> is already used for the help command.
- Meta-keys (with the ALT key pressed). Depending on your keyboard accented
  characters may be used as well. :map-alt-keys
- Use the '_' or ',' character and then any other character. The "_" and ","
  commands do exist in Vim (see _ and ,), but you probably never use them.
- Use a key that is a synonym for another command. For example: CTRL-P and
  CTRL-N. Use an extra character to allow more mappings.
- The key defined by <Leader> and one or more other keys. This is especially
  useful in scripts. mapleader

See the file "index" for keys that are not used and thus can be mapped without
losing any builtin function. You can also use ":help {key}^D" to find out if
a key is used for some command. ({key} is the specific key you want to find
out about, ^D is CTRL-D).

1.8 EXAMPLES

map-examples

A few examples (as you type them: for "<CR>" you type four characters).

```
:map <F3>  o#include
:map <M-g> /foo<CR>cwbar<Esc>
:map _x    d/END/e<CR>
:map! qq   quadrillion questions
```

Multiplying a count

When you type a count before triggering a mapping, it's like the count was
typed before the {lhs}. For example, with this mapping:

```
:map <F4>  3w
```

Typing 2<F4> will result in "23w". Thus not moving 2 * 3 words but 23 words.
If you want to multiply counts use the expression register:

```
:map <F4>  @='3w'<CR>
```

The part between quotes is the expression being executed. @=

1.9 USING MAPPINGS

map-typing

Vim will compare what you type with the start of a mapped sequence. If there
is an incomplete match, it will get more characters until there either is a
complete match or until there is no match at all. Example: If you map! "qq",
the first 'q' will not appear on the screen until you type another
character. This is because Vim cannot know if the next character will be a
'q' or not. If the 'timeout' option is on (which is the default) Vim will
only wait for one second (or as long as specified with the 'timeoutlen'
option). After that it assumes that the 'q' is to be interpreted as such. If
you type slowly, or your system is slow, reset the 'timeout' option. Then you
might want to set the 'ttimeout' option.

map-precedence

Buffer-local mappings (defined using :map-<buffer>) take precedence over
global mappings. When a buffer-local mapping is the same as a global mapping,
Vim will use the buffer-local mapping. In addition, Vim will use a complete
mapping immediately if it was defined with <nowait>, even if a longer mapping
has the same prefix. For example, given the following two mappings:

```
:map <buffer> <nowait> \a   :echo "Local \a"<CR>
:map                   \abc :echo "Global \abc"<CR>
```

When typing \a the buffer-local mapping will be used immediately. Vim will
not wait for more characters to see if the user might be typing \abc.

map-keys-fails

There are situations where key codes might not be recognized:

- Vim can only read part of the key code. Mostly this is only the first

  character. This happens on some Unix versions in an xterm.
- The key code is after character(s) that are mapped. E.g., "<F1><F1>" or

  "g<F1>".

The result is that the key code is not recognized in this situation, and the mapping fails. There are two actions needed to avoid this problem:

- Remove the 'K' flag from 'cpoptions'. This will make Vim wait for the rest

  of the characters of the function key.
- When using <F1> to <F4> the actual key code generated may correspond to

  <xF1> to <xF4>. There are mappings from <xF1> to <F1>, <xF2> to <F2>, etc.,
  but these are not recognized after another half a mapping. Make sure the
  key codes for <F1> to <F4> are correct:

```
:set <F1>=<type CTRL-V><type F1>
```

  Type the <F1> as four characters. The part after the "=" must be done with
  the actual keys, not the literal text.
Another solution is to use the actual key code in the mapping for the second special key:

```
:map <F1><Esc>OP :echo "yes"<CR>
```

Don't type a real <Esc>, Vim will recognize the key code and replace it with <F1> anyway.

recursive_mapping

If you include the {lhs} in the {rhs} you have a recursive mapping. When {lhs} is typed, it will be replaced with {rhs}. When the {lhs} which is included in {rhs} is encountered it will be replaced with {rhs}, and so on. This makes it possible to repeat a command an infinite number of times. The only problem is that the only way to stop this is by causing an error. The macros to solve a maze uses this, look there for an example. There is one exception: If the {rhs} starts with {lhs}, the first character is not mapped again (this is Vi compatible).
For example:

```
:map ab abcd
```

will execute the "a" command and insert "bcd" in the text. The "ab" in the {rhs} will not be mapped again.

If you want to exchange the meaning of two keys you should use the :noremap command. For example:

```
:noremap k j
:noremap j k
```

This will exchange the cursor up and down commands.

With the normal :map command mapping takes place until the text is found not to be a part of a {lhs}. For example, if you use:

```
:map x y
:map y x
```

Vim will replace x with y, and then y with x, etc. When this has happened 'maxmapdepth' times (default 1000), Vim will give the error message "recursive mapping".

:map-undo

If you include an undo command inside a mapped sequence, this will bring the text back in the state before executing the macro. This is compatible with the original Vi, as long as there is only one undo command in the mapped sequence (having two undo commands in a mapped sequence did not make sense in the original Vi, you would get back the text before the first undo).

1.10 MAPPING ALT-KEYS

:map-alt-keys

For a readable mapping command the <A-k> form can be used. Note that <A-k> and <A-K> are different, the latter will use an upper case letter. Actually, <A-K> and <A-S-K> are the same. Instead of "A" you can use "M". If you have an actual Meta modifier key, please see :map-meta-keys.

In the GUI Nvim handles the ALT key itself, thus mapping keys with ALT should always work. But in a terminal Nvim gets a sequence of bytes and has to figure out whether ALT was pressed. Terminals may use ESC to indicate that ALT was pressed. If ESC is followed by a {key} within 'ttimeoutlen'

milliseconds, the ESC is interpreted as:
        <ALT-{key}>
otherwise it is interpreted as two key presses:
        <ESC> {key}

1.11 MAPPING META-KEYS :map-meta-keys

Mapping keys with the Meta modifier works very similar to using the Alt key.
What key on your keyboard produces the Meta modifier depends on your keyboard
and configuration.

Note that mapping <M-a> actually is for using the Alt key. That can be
confusing! It cannot be changed, it would not be backwards compatible.

For the Meta modifier the "T" character is used. For example, to map Meta-b
in Insert mode:

```
:imap <T-b> terrible
```

1.12 MAPPING SUPER-KEYS or COMMAND-KEYS :map-super-keys :map-cmd-key

The Super / Command modifier is available if the terminal or GUI supports it.
The character "D" is used for the Super / Command modifier.

For example, to map Command-b in Insert mode:

```
:imap <D-b> barritone
```

1.13 MAPPING AN OPERATOR :map-operator

An operator is used before a {motion} command. To define your own operator
you must create a mapping that first sets the 'operatorfunc' option and then
invoke the g@ operator. After the user types the {motion} command the
specified function will be called.

g@ E774 E775

g@{motion} Call the function set by the 'operatorfunc' option.
            The '[mark is positioned at the start of the text
            moved over by {motion}, the '] mark on the last
            character of the text.
            The function is called with one String argument:
                "line"   {motion} was linewise
                "char"   {motion} was charwise
                "block"   {motion} was blockwise-visual
            The type can be forced, see forced-motion.

Here is an example that counts the number of spaces with <F4>:

```
nnoremap <expr> <F4> CountSpaces()
xnoremap <expr> <F4> CountSpaces()
" doubling <F4> works on a line
nnoremap <expr> <F4><F4> CountSpaces() .. '_'

function CountSpaces(context = {}, type = '') abort
  if a:type == ''
    let context = #{
      \ dot_command: v:false,
      \ extend_block: '',
      \ virtualedit: [&l:virtualedit, &g:virtualedit],
      \ }
    let &operatorfunc = function('CountSpaces', [context])
    set virtualedit=block
    return 'g@'
  endif

  let save = #{
    \ clipboard: &clipboard,
    \ selection: &selection,
    \ virtualedit: [&l:virtualedit, &g:virtualedit],
    \ register: getreginfo('"'),
    \ visual_marks: [getpos("'<"), getpos("'>")],
    \ }

  try
    set clipboard= selection=inclusive virtualedit=
    let commands = #{
      \ line: "'[V']",
      \ char: "`[v`]",
      \ block: "`[\<C-V>`]",
      \ }[a:type]
    let [_, _, col, off] = getpos("']")
    if off != 0
      let vcol = getline("'[")->strpart(0, col + off)->strdisplaywidth()
      if vcol >= [line("'["), '$']->virtcol() - 1
        let a:context.extend_block = '$'
      else
        let a:context.extend_block = vcol .. '|'
      endif
    endif
    if a:context.extend_block != ''
      let commands ..= 'oO' .. a:context.extend_block
    endif
    let commands ..= 'y'
    execute 'silent noautocmd keepjumps normal! ' .. commands
    echomsg getreg('"')->count(' ')
  finally
    call setreg('"', save.register)
    call setpos("'<", save.visual_marks[0])
    call setpos("'>", save.visual_marks[1])
    let &clipboard = save.clipboard
    let &selection = save.selection
    let [&l:virtualedit, &g:virtualedit] = get(a:context.dot_command ? save : a:context, 'virtualedit')
    let a:context.dot_command = v:true
  endtry
endfunction
```

An <expr> mapping is used to be able to fetch any prefixed count and register.
This also avoids using a command line, which would trigger CmdlineEnter and
CmdlineLeave autocommands.

Note that the 'selection' option is temporarily set to "inclusive" to be able
to yank exactly the right text by using Visual mode from the '[to the ']
mark.

Also note that the 'clipboard' option is temporarily emptied to avoid
clobbering the "* or "+ registers, if its value contains the item unnamed
or unnamedplus.

The mode() function will return the state as it will be after applying the
operator.

Here is an example for using a lambda function to create a normal-mode
operator to add quotes around text in the current line:

```
nnoremap <F4> <Cmd>let &opfunc='{t ->
                        \ getline(".")
                        \ ->split("\\zs")
                        \ ->insert("\"", col("']"))
                        \ ->insert("\"", col("'[") - 1)
                        \ ->join("")
                        \ ->setline(".")}'<CR>g@
```

2. Abbreviations abbreviation abbreviations Abbreviations

Abbreviations are used in Insert mode, Replace mode and Command-line mode.
If you enter a word that is an abbreviation, it is replaced with the word it

stands for. This can be used to save typing for often used long words. And
you can use it to automatically correct obvious spelling errors.
Examples:

```
        :iab ms Microsoft
        :iab tihs this
```

There are three types of abbreviations:

full-id   The "full-id" type consists entirely of keyword characters (letters
          and characters from 'iskeyword' option). This is the most common
          abbreviation.

          Examples: "foo", "g3", "-1"

end-id   The "end-id" type ends in a keyword character, but all the other
          characters are not keyword characters.

          Examples: "#i", "..f", "$/7"

non-id   The "non-id" type ends in a non-keyword character, the other
          characters may be of any type, excluding space and tab.

          Examples: "def#", "4/7$"

Examples of strings that cannot be abbreviations: "a.b", "#def", "a b", "_$r"

An abbreviation is only recognized when you type a non-keyword character.
This can also be the <Esc> that ends Insert mode or the <CR> that ends a
command. The non-keyword character which ends the abbreviation is inserted
after the expanded abbreviation. An exception to this is the character <C-]>,
which is used to expand an abbreviation without inserting any extra
characters.

Example:

```
:ab hh  hello
```

```
            "hh<Space>" is expanded to "hello<Space>"
            "hh<C-]>" is expanded to "hello"
```

The characters before the cursor must match the abbreviation. Each type has
an additional rule:

full-id   In front of the match is a non-keyword character, or this is where
          the line or insertion starts. Exception: When the abbreviation is
          only one character, it is not recognized if there is a non-keyword
          character in front of it, other than a space or a tab. However, for
          the command line "'<,'>" (or any other marks) is ignored, as if the
          command line starts after it.

end-id   In front of the match is a keyword character, or a space or a tab,
          or this is where the line or insertion starts.

non-id   In front of the match is a space, tab or the start of the line or
          the insertion.

Examples: ({CURSOR} is where you type a non-keyword character)

```
:ab foo   four old otters
```

```
                " foo{CURSOR}"    is expanded to " four old otters"
                " foobar{CURSOR}" is not expanded
                "barfoo{CURSOR}"  is not expanded
```

```
:ab #i #include
```

```
                "#i{CURSOR}"      is expanded to "#include"
                ">#i{CURSOR}"     is not expanded
```

```
:ab ;; <endofline>
```

```
                "test;;"          is not expanded
                "test ;;"         is expanded to "test <endofline>"
```

To avoid the abbreviation in Insert mode: Type CTRL-V before the character
that would trigger the abbreviation. E.g. CTRL-V <Space>. Or type part of
the abbreviation, exit insert mode with <Esc>, re-enter insert mode with "a"
and type the rest.

To avoid the abbreviation in Command-line mode: Type CTRL-V twice somewhere in
the abbreviation to avoid it to be replaced. A CTRL-V in front of a normal
character is mostly ignored otherwise.

It is possible to move the cursor after an abbreviation:

```
:iab if if ()<Left>
```

You can even do more complicated things. For example, to consume the space
typed after an abbreviation:

```
func Eatchar(pat)
   let c = nr2char(getchar(0))
   return (c =~ a:pat) ? '' : c
endfunc
iabbr <silent> if if ()<Left><C-R>=Eatchar('\s')<CR>
```

There are no default abbreviations.

Abbreviations are never recursive. You can use ":ab f f-o-o" without any
problem. But abbreviations can be mapped.

:abbreviate-local :abbreviate-<buffer>
Just like mappings, abbreviations can be local to a buffer. This is mostly
used in a filetype-plugin file. Example for a C plugin file:

```
:abb <buffer> FF  for (i = 0; i < ; ++i)
```

:ab :abbreviate
:ab[breviate]   list all abbreviations. The character in the first
                        column indicates the mode where the abbreviation is
                        used: 'i' for insert mode, 'c' for Command-line
                        mode, '!' for both. These are the same as for
                        mappings, see map-listing.

:abbreviate-verbose

When 'verbose' is non-zero, listing an abbreviation will also display where it
was last defined. Example:

```
:verbose abbreviate
!  teh           the
        Last set from /home/abcd/vim/abbr.vim
```

See :verbose-cmd for more information.

:ab[breviate] {lhs}   list the abbreviations that start with {lhs}
                        You may need to insert a CTRL-V (type it twice) to
                        avoid that a typed {lhs} is expanded, since
                        command-line abbreviations apply here.

:ab[breviate] [<expr>] [<buffer>] {lhs} {rhs}
                        add abbreviation for {lhs} to {rhs}. If {lhs} already
                        existed it is replaced with the new {rhs}. {rhs} may
                        contain spaces.
                        See :map-<expr> for the optional <expr> argument.
                        See :map-<buffer> for the optional <buffer> argument.

:una :unabbreviate
:una[bbreviate] [<buffer>] {lhs}
                        Remove abbreviation for {lhs} from the list. If none
                        is found, remove abbreviations in which {lhs} matches
                        with the {rhs}. This is done so that you can even
                        remove abbreviations after expansion. To avoid
                        expansion insert a CTRL-V (type it twice).

:norea :noreabbrev
:norea[bbrev] [<expr>] [<buffer>] [lhs] [rhs]
                        same as ":ab", but no remapping for this {rhs}

:ca :cab :cabbrev
:ca[bbrev] [<expr>] [<buffer>] [lhs] [rhs]
                        same as ":ab", but for Command-line mode only.

:cuna :cunabbrev
:cuna[bbrev] [<buffer>] {lhs}
                        Same as ":una", but for Command-line mode only.

:cnorea :cnoreabbrev
:cnorea[bbrev] [<expr>] [<buffer>] [lhs] [rhs]
                        same as ":ab", but for Command-line mode only and no
                        remapping for this {rhs}

:ia :iabbrev
:ia[bbrev] [<expr>] [<buffer>] [lhs] [rhs]
                        same as ":ab", but for Insert mode only.

:iuna :iunabbrev
:iuna[bbrev] [<buffer>] {lhs}
                        Same as ":una", but for insert mode only.

:inorea :inoreabbrev
:inorea[bbrev] [<expr>] [<buffer>] [lhs] [rhs]
                        same as ":ab", but for Insert mode only and no
                        remapping for this {rhs}

:abc :abclear
:abc[lear] [<buffer>]   Remove all abbreviations.

:iabc :iabclear
:iabc[lear] [<buffer>] Remove all abbreviations for Insert mode.

:cabc :cabclear
:cabc[lear] [<buffer>] Remove all abbreviations for Command-line mode.

using_CTRL-V

It is possible to use special characters in the rhs of an abbreviation.
CTRL-V has to be used to avoid the special meaning of most non printable
characters. How many CTRL-Vs need to be typed depends on how you enter the
abbreviation. This also applies to mappings. Let's use an example here.

Suppose you want to abbreviate "esc" to enter an <Esc> character. When you
type the ":ab" command in Vim, you have to enter this: (here ^V is a CTRL-V
and ^[is <Esc>)

You type: ab esc ^V^V^V^V^[

All keyboard input is subjected to ^V quote interpretation, so the first, third, and fifth ^V characters simply allow the second, and fourth ^Vs, and the ^[, to be entered into the command-line.

You see: ab esc ^V^V^[

The command-line contains two actual ^Vs before the ^[. This is how it should appear in your vimrc file, if you choose to go that route. The first ^V is there to quote the second ^V; the :ab command uses ^V as its own quote character, so you can include quoted whitespace or the | character in the abbreviation. The :ab command doesn't do anything special with the ^[character, so it doesn't need to be quoted. (Although quoting isn't harmful; that's why typing 7 [but not 8!] ^Vs works.)

Stored as: esc ^V^[

After parsing, the abbreviation's short form ("esc") and long form (the two characters "^V^[") are stored in the abbreviation table. If you give the :ab command with no arguments, this is how the abbreviation will be displayed.

Later, when the abbreviation is expanded because the user typed in the word "esc", the long form is subjected to the same type of ^V interpretation as keyboard input. So the ^V protects the ^[character from being interpreted as the "exit Insert mode" character. Instead, the ^[is inserted into the text.

Expands to: ^[

[example given by Steve Kirkendall]

3. Local mappings and functions script-local

When using several Vim script files, there is the danger that mappings and functions used in one script use the same name as in other scripts. To avoid this, they can be made local to the script.

<SID> <SNR> E81

The string "<SID>" can be used in a mapping or menu. This is useful if you have a script-local function that you want to call from a mapping in the same script.

When executing the map command, Vim will replace "<SID>" with the special key code <SNR>, followed by a number that's unique for the script, and an underscore. Example:

```
:map <SID>Add
```

would define a mapping "<SNR>23_Add".

When defining a function in a script, "s:" can be prepended to the name to make it local to the script. But when a mapping is executed from outside of the script, it doesn't know in which script the function was defined. To avoid this problem, use "<SID>" instead of "s:". The same translation is done as for mappings. This makes it possible to define a call to the function in a mapping.

When a local function is executed, it runs in the context of the script it was defined in. This means that new functions and mappings it defines can also use "s:" or "<SID>" and it will use the same unique number as when the function itself was defined. Also, the "s:var" local script variables can be used.

When executing an autocommand or a user command, it will run in the context of the script it was defined in. This makes it possible that the command calls a local function or uses a local mapping.

In case the value is used in a context where <SID> cannot be correctly expanded, use the expand() function:

```
let &includexpr = expand('<SID>') .. 'My_includeexpr()'
```

Otherwise, using "<SID>" outside of a script context is an error.

If you need to get the script number to use in a complicated script, you can use this function:

```
func s:ScriptNumber()
  return matchstr(expand('<SID>'), '<SNR>\zs\d\+\ze_')
endfunc
```

The "<SNR>" will be shown when listing functions and mappings. This is useful to find out what they are defined to.

The :scriptnames command can be used to see which scripts have been sourced and what their <SNR> number is.

4. User-defined commands user-commands

It is possible to define your own Ex commands. A user-defined command can act just like a built-in command (it can have a range or arguments, arguments can be completed as filenames or buffer names, etc), except that when the command is executed, it is transformed into a normal Ex command and then executed.

For starters: See section 40.2 in the user manual.

E183 E841 user-cmd-ambiguous

All user defined commands must start with an uppercase letter, to avoid confusion with builtin commands. Exceptions are these builtin commands:

```
:Next
```

They cannot be used for a user defined command.

The other characters of the user command can be uppercase letters, lowercase letters or digits. When using digits, note that other commands that take a numeric argument may become ambiguous. For example, the command ":Cc2" could be the user command ":Cc2" without an argument, or the command ":Cc" with argument "2". It is advised to put a space between the command name and the argument to avoid these problems.

When using a user-defined command, the command can be abbreviated. However, if an abbreviation is not unique, an error will be issued. Furthermore, a built-in command will always take precedence.

Example:

```
:command Rename ...
:command Renumber ...
:Rena                           " Means "Rename"
:Renu                           " Means "Renumber"
:Ren                            " Error - ambiguous
:command Paste ...
```

It is recommended that full names for user-defined commands are used in scripts.

:com[mand] :com :command
List all user-defined commands. When listing commands, the characters in the first columns are:

```
    !   Command has the -bang attribute
    "   Command has the -register attribute
    |   Command has the -bar attribute
    b   Command is local to current buffer
```

(see below for details on attributes)
The list can be filtered on command name with :filter, e.g., to list all commands with "Pyth" in the name:

```
filter Pyth command
```

:com[mand] {cmd} List the user-defined commands that start with {cmd}

:command-verbose

When 'verbose' is non-zero, listing a command will also display where it was last defined and any completion argument. Example:

```
:verbose command TOhtml
```

Name	Args	Range	Complete	Definition
TOhtml	0	%		:call Convert2HTML(<line1>, <line2>)
Last set from /usr/share/vim/vim-7.0/plugin/tohtml.vim				

See :verbose-cmd for more information.

E174 E182

:com[mand][!] [{attr}...] {cmd} {repl}
Define a user command. The name of the command is {cmd} and its replacement text is {repl}. The command's attributes (see below) are {attr}. If the command already exists, an error is reported, unless a ! is specified, in which case the command is redefined. There is one exception: When sourcing a script again, a command that was previously defined in that script will be silently replaced.

:delc[ommand] {cmd} :delc :delcommand E184
Delete the user-defined command {cmd}.

:delc[ommand] -buffer {cmd} E1237

Delete the user-defined command {cmd} that was defined for the current buffer.

:comc[lear] :comc :comclear
Delete all user-defined commands.

Command attributes

command-attributes

User-defined commands are treated by Nvim just like any other Ex commands. They can have arguments, or have a range specified. Arguments are subject to completion as filenames, buffers, etc. Exactly how this works depends upon the command's attributes, which are specified when the command is defined.

When defining a user command in a script, it will be able to call functions local to the script and use mappings local to the script. When the user invokes the user command, it will run in the context of the script it was defined in. This matters if <SID> is used in a command.

There are a number of attributes, split into four categories: argument handling, completion behavior, range handling, and special cases. The attributes are described below, by category.

Argument handling

E175 E176 :command-nargs

By default, a user defined command will take no arguments (and an error is reported if any are supplied). However, it is possible to specify that the command can take arguments, using the -nargs attribute. Valid cases are:

```
        -nargs=0    No arguments are allowed (the default)
        -nargs=1    Exactly one argument is required, it includes spaces;
                    completion treats white spaces as argument separation
        -nargs=_    Exactly one argument is required, it includes spaces;
                    completion treats white spaces as part of the argument
        -nargs=*    Any number of arguments are allowed (0, 1, or many),
                    separated by white space
        -nargs=?    0 or 1 arguments are allowed
        -nargs=+    Arguments must be supplied, but any number are allowed
```

Arguments are considered to be separated by (unescaped) spaces or tabs in this context, except when there is one argument, then the white space is part of the argument. The difference between the "-nargs=1" and "-nargs=_":

```
func MyComplete(ArgLead, CmdLine, CursorPos)
  return ["one value", "two values", "three values"]
        \->matchfuzzy(a:ArgLead)
endfunc
:command -nargs=1 -complete=customlist,MyComplete MyCmd1 echo <q-args>
:command -nargs=_ -complete=customlist,MyComplete MyCmd2 echo <q-args>
```

Completing ":MyCmd1 two va<tab>" will complete with:

```
:MyCmd1 two one value
```

Completing ":MyCmd2 two va<tab>" will complete with:

```
:MyCmd2 two values
```

Note that arguments are used as text, not as expressions. Specifically, "s:var" will use the script-local variable in the script where the command was defined, not where it is invoked! Example:

script1.vim:

```
:let s:error = "None"
:command -nargs=1 Error echoerr <args>
```

script2.vim:

```
:source script1.vim
:let s:error = "Wrong!"
:Error s:error
```

Executing script2.vim will result in "None" being echoed. Not what you intended! Calling a function may be an alternative.

Completion behavior

:command-completion E179 E180 E181

:command-complete

By default, the arguments of user defined commands do not undergo completion. However, by specifying one or the other of the following attributes, argument completion can be enabled:

```
        -complete=arglist       file names in argument list
        -complete=augroup       autocmd groups
        -complete=breakpoint    :breakadd suboptions
        -complete=buffer        buffer names
        -complete=color         color schemes
        -complete=command       Ex command (and arguments)
        -complete=compiler      compilers
        -complete=diff_buffer   diff buffer names
        -complete=dir           directory names
        -complete=dir_in_path   directory names in 'cdpath'
        -complete=environment   environment variable names
        -complete=event         autocommand events
        -complete=expression    Vim expression
        -complete=file          file and directory names
        -complete=file_in_path  file and directory names in 'path'
        -complete=filetype      filetype names 'filetype'
        -complete=function      function name
        -complete=help          help subjects
        -complete=highlight     highlight groups
        -complete=history       :history suboptions
        -complete=keymap        keyboard mappings
        -complete=locale        locale names (as output of locale -a)
        -complete=lua           Lua expression :lua
        -complete=mapclear      buffer argument
```

```
        -complete=mapping       mapping name
        -complete=menu          menus
        -complete=messages      :messages suboptions
        -complete=option        options
        -complete=packadd       optional package pack-add names
        -complete=retab         :retab suboptions
        -complete=runtime       file and directory names in 'runtimepath'
        -complete=scriptnames   sourced script names
        -complete=shellcmd      Shell command
        -complete=shellcmdline  First is a shell command and subsequent ones
                                are filenames.  The same behavior as :!cmd.
                                To get correct completion, :command-nargs
                                should be '*' or '+'
        -complete=sign          :sign suboptions
        -complete=syntax        syntax file names 'syntax'
        -complete=syntime       :syntime suboptions
        -complete=tag           tags
        -complete=tag_listfiles tags, file names are shown when CTRL-D is hit
        -complete=user          user names
        -complete=var           user variables
        -complete=custom,{func} custom completion, defined via {func}
        -complete=customlist,{func} custom completion, defined via {func}
```

If you specify completion while there is nothing to complete (-nargs=0, the default) then you get error E1208 .
Note: That some completion methods might expand environment variables.

Custom completion

:command-completion-custom

:command-completion-customlist E467 E468
It is possible to define customized completion schemes via the "custom,{func}" or the "customlist,{func}" completion argument. The {func} part should be a function with the following signature:

```
:function {func}(ArgLead, CmdLine, CursorPos)
```

The function need not use all these arguments. The function should provide the completion candidates as the return value.

For the "custom" argument, the function should return the completion candidates one per line in a newline separated string.

For the "customlist" argument, the function should return the completion candidates as a Vim List. Each item may be either a string or a Dictionary. A Dictionary item may have the following keys:

```
        word    (required) the text inserted into the command line when the
                item is selected
        abbr    alternative text shown in the popup menu in place of "word",
                when 'wildoptions' contains "pum"; useful when the inserted
                text and the displayed text should differ
        kind    short kind text (one or two characters), shown in the popup
                menu when 'wildoptions' contains "pum"
        menu    extra text shown after the match in the popup menu
        info    long description shown in the info popup
```

Items that are neither a string nor a Dictionary, and Dictionary items without a "word" key, are ignored. When 'wildoptions' does not contain "pum", only "word" is shown.

The function arguments are:

```
        ArgLead         the leading portion of the argument currently being
                        completed on; note that this only captures the current
                        space-separated word, even when using "-nargs=1"
        CmdLine         the entire command line
        CursorPos       the cursor position in it (byte index)
```

The function may use these for determining context. For the "custom" argument, it is not necessary to filter candidates against the (implicit pattern in) ArgLead. Vim will filter the candidates with its regexp engine after function return, and this is probably more efficient in most cases. If 'wildoptions' contains "fuzzy", then the candidates will be filtered using fuzzy-matching. For the "customlist" argument, Vim will not filter the returned completion candidates and the user supplied function should filter the candidates.

The following example lists user names to a Finger command

```
:com -complete=custom,ListUsers -nargs=1 Finger !finger <args>
:fun ListUsers(A,L,P)
:    return system("cut -d: -f1 /etc/passwd")
:endfun
```

The following example completes filenames from the directories specified in the 'path' option:

```
:com -nargs=1 -bang -complete=customlist,EditFileComplete
                    \ EditFile edit<bang> <args>
:fun EditFileComplete(A,L,P)
:    return split(globpath(&path, a:A), "\n")
:endfun
```

This example does not work for file names with spaces!

Range handling

E177 E178 :command-range :command-count

By default, user-defined commands do not accept a line number range. However, it is possible to specify that the command does take a range (the -range attribute), or that it takes an arbitrary count value, either in the line number position (-range=N, like the :split command) or as a "count" argument (-count=N, like the :Next command). The count will then be available in the argument with <count>.

Possible attributes are:

```
        -range      Range allowed, default is current line
        -range=%    Range allowed, default is whole file (1,$)
        -range=N    A count (default N) which is specified in the line
                    number position (like :split); allows for zero line
                    number.
        -count=N    A count (default N) which is specified either in the line
                    number position, or as an initial argument (like :Next).
        -count      Acts like -count=0
```

Note that -range=N and -count=N are mutually exclusive - only one should be specified.

:command-addr

It is possible that the special characters in the range like ., $ or % which by default correspond to the current line, last line and the whole buffer, relate to arguments, (loaded) buffers, windows or tabpages.

Possible values are (second column is the short name used in listing):

```
    -addr=lines                 Range of lines (the default for -range)
    -addr=arguments       arg   Range for arguments
    -addr=buffers         buf   Range for buffers (also not loaded buffers)
    -addr=loaded_buffers  load  Range for loaded buffers
    -addr=windows         win   Range for windows
    -addr=tabs            tab   Range for tabpages
    -addr=quickfix        qf    Range for quickfix entries
    -addr=other           ?     Other kind of range; can use ".", "$" and "%"
                                as with "lines" (the default for -count)
```

Incremental preview

:command-preview {nvim-api}

Commands can show an 'inccommand' (as-you-type) preview by defining a preview handler (only from Lua, see nvim_create_user_command()).

Before the preview callback is executed, Nvim will temporarily disable 'cursorline' and 'cursorcolumn' to avoid highlighting issues.

The preview callback must be a Lua function with this signature:

```
function cmdpreview(opts, ns, buf)
```

where "opts" has the same form as that given to nvim_create_user_command() callbacks, "ns" is the preview namespace id for highlights, and "buf" is the buffer that your preview routine will directly modify to show the previewed results (for "inccommand=split", or nil for "inccommand=nosplit").

Your command preview routine must implement this protocol:

1. Modify the target buffers as required for the preview (see nvim_buf_set_text() and nvim_buf_set_lines()).
2. If preview buffer is provided, add necessary text to the preview buffer.
3. Add required highlights to the target buffers. If preview buffer is provided, add required highlights to the preview buffer as well. All highlights must be added to the preview namespace which is provided as an argument to the preview callback (see vim.hl.range() and nvim_buf_set_extmark() for help on how to add highlights to a namespace).
4. Return an integer (0, 1, 2) which controls how Nvim behaves as follows:

   0: No preview is shown.
   1: Preview is shown without preview window (even with "inccommand=split").
   2: Preview is shown and preview window is opened (if "inccommand=split").
      For "inccommand=nosplit" this is the same as 1.

After preview ends, Nvim discards all changes to all buffers made during the preview and clears all highlights in the preview namespace.

Here's an example of a command to trim trailing whitespace from lines that supports incremental command preview:

```
-- If invoked as a preview callback, performs 'inccommand' preview by
-- highlighting trailing whitespace in the current buffer.
local function trim_space_preview(opts, preview_ns, preview_buf)
  vim.cmd('hi clear Whitespace')
  local line1 = opts.line1
  local line2 = opts.line2
  local buf = vim.api.nvim_get_current_buf()
  local lines = vim.api.nvim_buf_get_lines(buf, line1 - 1, line2, false)
  local preview_buf_line = 0

  for i, line in ipairs(lines) do
    local start_idx, end_idx = string.find(line, '%s+$')

    if start_idx then
      -- Highlight the match
      vim.hl.range(
        buf,
        preview_ns,
        'Substitute',
        {line1 + i - 2, start_idx - 1},
        {line1 + i - 2, end_idx}
      )

      -- Add lines and set highlights in the preview buffer
      -- if inccommand=split
      if preview_buf then
        local prefix = string.format('|%d| ', line1 + i - 1)

        vim.api.nvim_buf_set_lines(
          preview_buf,
          preview_buf_line,
          preview_buf_line,
          false,
          { prefix .. line }
        )
        vim.hl.range(
          preview_buf,
          preview_ns,
          'Substitute',
          {preview_buf_line, #prefix + start_idx - 1},
          {preview_buf_line, #prefix + end_idx}
        )
        preview_buf_line = preview_buf_line + 1
      end
    end
  end

  -- Return the value of the preview type
  return 2
end

-- Trims all trailing whitespace in the current buffer.
local function trim_space(opts)
  local line1 = opts.line1
  local line2 = opts.line2
  local buf = vim.api.nvim_get_current_buf()
  local lines = vim.api.nvim_buf_get_lines(buf, line1 - 1, line2, false)

  local new_lines = {}
  for i, line in ipairs(lines) do
    new_lines[i] = string.gsub(line, '%s+$', '')
  end
  vim.api.nvim_buf_set_lines(buf, line1 - 1, line2, false, new_lines)
end

-- Create the user command
vim.api.nvim_create_user_command(
  'TrimTrailingWhitespace',
  trim_space,
  { nargs = '?', range = '%', addr = 'lines', preview = trim_space_preview }
)
```

Special cases

:command-bang :command-bar
:command-register :command-buffer
:command-keepscript

There are some special cases as well:

```
        -bang       The command can take a ! modifier (like :q or :w)
        -bar        The command can be followed by a "|" and another command.
                    A "|" inside the command argument is not allowed then.
                    Also checks for a " to start a comment.
        -register   The first argument to the command can be an optional
                    register name (like :del, :put, :yank).
        -buffer     The command will only be available in the current buffer.
        -keepscript Do not use the location of where the user command was
                    defined for verbose messages, use the location of where
                    the user command was invoked.
```

In the cases of the -count and -register attributes, if the optional argument is supplied, it is removed from the argument list and is available to the replacement text separately.
Note that these arguments can be abbreviated, but that is a deprecated feature. Use the full name for new scripts.

Replacement text

The replacement text {repl} for a user defined command is scanned for special escape sequences, using <...> notation. Escape sequences are replaced with values from the entered command line, and all other text is copied unchanged. The resulting string is executed as an Ex command. To avoid the replacement use <lt> in place of the initial <. Thus to include "<bang>" literally use "<lt>bang>".

The valid escape sequences are

<line1>

<line1> The starting line of the command range.

<line2>

<line2> The final line of the command range.

<range>

<range> The number of items in the command range: 0, 1 or 2

<count>

<count> Any count supplied (as described for the '-range' and '-count' attributes).

<bang>

<bang> (See the '-bang' attribute) Expands to a ! if the command was executed with a ! modifier, otherwise expands to nothing.

<mods> <q-mods> :command-modifiers

<mods> The command modifiers, if specified. Otherwise, expands to nothing. Supported modifiers are :aboveleft, :belowright, :botright, :browse, :confirm, :hide, :horizontal, :keepalt, :keepjumps, :keepmarks, :keeppatterns, :leftabove, :lockmarks, :noautocmd, :noswapfile, :rightbelow, :sandbox, :silent, :tab, :topleft, :unsilent, :verbose, and :vertical.
Note that :filter is not supported.
Examples:

```
command! -nargs=+ -complete=file MyEdit
            \ for f in expand(<q-args>, 0, 1) |
            \ exe '<mods> split ' .. f |
            \ endfor

function! SpecialEdit(files, mods)
    for f in expand(a:files, 0, 1)
        exe a:mods .. ' split ' .. f
    endfor
endfunction
command! -nargs=+ -complete=file Sedit
            \ call SpecialEdit(<q-args>, <q-mods>)
```

<reg> <register>

<reg>   (See the '-register' attribute) The optional register, if specified. Otherwise, expands to nothing. <register> is a synonym for this.

<args>

<args> The command arguments, exactly as supplied (but as noted above, any count or register can consume some of the arguments, which are then not part of <args>).

<lt>   A single '<' (Less-Than) character. This is needed if you want to get a literal copy of one of these escape sequences into the expansion - for example, to get <bang>, use <lt>bang>.

<q-args>

If the first two characters of an escape sequence are "q-" (for example, <q-args>) then the value is quoted in such a way as to make it a valid value for use in an expression. This uses the argument as one single value. When there is no argument <q-args> is an empty string. See the q-args-example below.

<f-args>

To allow commands to pass their arguments on to a user-defined function, there is a special form <f-args> ("function args"). This splits the command arguments at spaces and tabs, quotes each argument individually, and the <f-args> sequence is replaced by the comma-separated list of quoted arguments. See the Mycmd example below. If no arguments are given <f-args> is removed.
   To embed whitespace into an argument of <f-args>, prepend a backslash. <f-args> replaces every pair of backslashes (\\) with one backslash. A backslash followed by a character other than white space or a backslash remains unmodified. Also see f-args-example below. Overview:

command	<f-args>
XX ab	"ab"
XX a\b	'a\b'
XX a\ b	'a b'
XX a\ b	'a ', 'b'
XX a\\b	'a\b'
XX a\\ b	'a\', 'b'
XX a\\\b	'a\\b'
XX a\\\ b	'a\ b'
XX a\\\\b	'a\\b'
XX a\\\\ b	'a\\', 'b'
XX	[nothing]

Note that if the "no arguments" situation is to be handled, you have to make
sure that the function can be called without arguments.

Examples for user commands:

```
" Delete everything after here to the end
:com Ddel +,$d

" Rename the current buffer
:com -nargs=1 -bang -complete=file Ren f <args>|w<bang>

" Replace a range with the contents of a file
" (Enter this all as one line)
:com -range -nargs=1 -complete=file
      Replace <line1>-pu_|<line1>,<line2>d|r <args>|<line1>d

" Count the number of lines in the range
:com! -range -nargs=0 Lines  echo <line2> - <line1> + 1 "lines"
```

f-args-example

Call a user function (example of <f-args>)

```
:com -nargs=* Mycmd call Myfunc(<f-args>)
```

When executed as:

```
:Mycmd arg1 arg2
```

This will invoke:

```
:call Myfunc("arg1","arg2")
```

q-args-example

A more substantial example:

```
:function Allargs(command)
:   let i = 0
:   while i < argc()
:         if filereadable(argv(i))
:            execute "e " .. argv(i)
:            execute a:command
:      endif
:      let i = i + 1
:   endwhile
:endfunction
:command -nargs=+ -complete=command Allargs call Allargs(<q-args>)
```

The command Allargs takes any Vim command(s) as argument and executes it on
all files in the argument list. Usage example (note use of the "e" flag to
ignore errors and the "update" command to write modified buffers):

```
:Allargs %s/foo/bar/ge|update
```

This will invoke:

```
:call Allargs("%s/foo/bar/ge|update")
```

Notable changes since Nvim 0.9

For changes in the previous release, see news-0.9.

BREAKING CHANGES

These changes may require adaptations in your config or plugins.

- API:
  - nvim_open_win() now blocks all autocommands when `noautocmd` is set, rather than just those from setting the `buffer` to display in the window.
- Defaults:
  - Default color scheme has been updated to be "Nvim branded" and accessible. Use `:colorscheme vim` to revert to the old legacy color scheme.
    - These Nvim specific highlight groups are now defined in a meaningfully different way and might need an update:
      - hl-FloatBorder is linked to hl-NormalFloat instead of hl-WinSeparator.
      - hl-NormalFloat is not linked to hl-Pmenu.
      - hl-WinBar has different background.
      - hl-WinBarNC is similar to hl-WinBar but not bold.
      - hl-WinSeparator is linked to hl-Normal instead of hl-VertSplit.
    - This also might result into some color schemes looking differently due to them relying on implicit assumptions about how highlight groups are defined by default. To account for this, define all attributes of highlight groups explicitly. Alternatively, use `:colorscheme vim` or `:source $VIMRUNTIME/colors/vim.lua` to restore previous definitions.
  - 'termguicolors' is enabled by default when Nvim is able to determine that the host terminal emulator supports 24-bit color.
- Editor:
  - When switching windows, CursorMoved autocommands trigger when Nvim is back on the main loop rather than immediately. This is more compatible with Vim.
  - "#" followed by a digit no longer stands for a function key at the start of the lhs of a mapping.
  - shm-q now fully hides macro recording message instead of only shortening it.
  - Signs placed through the legacy sign-commands are now stored and displayed as extmarks internally. Along with the following changes:
    - A sign placed twice in the same group with the same identifier will be moved.
    - Legacy signs are always deleted along with the line it is placed on.
    - Legacy and extmark signs will show up in both :sign-place-list and nvim_buf_get_extmarks().
    - Legacy and extmark signs are displayed and listed with the same priority: line number -> priority -> sign id -> recently placed
  - `:behave` was removed.
    - If you used `:behave xterm`, the following is equivalent:

      ```
      set mousemodel=extend
      ```

    - If you used `:behave mswin`, the following is equivalent:

      ```
      set selection=exclusive
      set selectmode=mouse,key
      set mousemodel=popup
      set keymodel=startsel,stopsel
      ```

- Events:
  - Returning any truthy value from a callback passed to nvim_create_autocmd() (rather than just `true`) will delete the autocommand.
- LSP:
  - LanguageTree:parse() will no longer parse injections by default and now requires an explicit range argument to be passed. If injections are required, provide an explicit range via `parser:parse({ start_row, end_row })`.
  - vim.lsp.util.parse_snippet() will now strictly follow the snippet grammar defined by LSP, and hence previously parsed snippets might now be considered invalid input.
  - vim.lsp.codelens.refresh() now takes an `opts` argument. With this change, the default behavior of just refreshing the current buffer has been replaced by refreshing all buffers.
  - vim.lsp.util.extract_completion_items() will no longer return reliable results, since it does not apply `itemDefaults` when its input is a `CompletionList`. Moreover, since support for LSP `completionList.itemDefaults` was added, some third party plugins might be negatively impacted in case the language servers support the feature but the plugin does not. If necessary, the respective capability can be removed when calling vim.lsp.protocol.make_client_capabilities().
  - LspRequest and LspProgressUpdate (renamed to LspProgress) autocmds were promoted from User autocmds to first class citizens.
- Lua:
  - -l ensures output ends with a newline if the script prints messages and doesn't cause Nvim to exit.
  - Removed functions from the vim.json module:
    - Unnecessary, undocumented functions which caused global side-effects.
    - `vim.json.null` is redundant with `vim.NIL`.
    - `vim.json.array_mt` (and related) is redundant with `vim.empty_dict()`.
  - vim.islist() now checks whether a table is actually list-like (i.e., has integer keys without gaps and starting from 1). For the previous behavior (only check for integer keys, allow gaps or not starting with 1), use vim.isarray().
  - Renamed `vim.treesitter.playground` to `vim.treesitter.dev`.
- Options:
  - Removed some Vim 5.0<= option compatibilities:
    - 'backspace' no longer supports number values. Instead:
      - for backspace=0 set backspace= (empty)
      - for backspace=1 set backspace=indent,eol
      - for backspace=2 set backspace=indent,eol,start (default behavior in Nvim)
      - for backspace=3 set backspace=indent,eol,nostop

    - 'backupdir' and 'directory' will no longer remove a > at the start of the option.
  - OptionSet autocommand args v:option_new, v:option_old, v:option_oldlocal, v:option_oldglobal now have the type of the option instead of always being strings. v:option_old is now the old global value for all global-local options, instead of just string global-local options.
  - Local value for a global-local number/boolean option is now unset when the option is set (e.g. using :set or nvim_set_option_value()) without a scope, which means they now behave the same way as string options.
- Plugins:
  - :TOhtml has been rewritten in Lua to support Nvim-specific decorations, and many options have been removed.
- Treesitter:
  - Treesitter highlight groups have been renamed to be more in line with upstream tree-sitter and Helix to make it easier to share queries. The full list is documented in treesitter-highlight-groups.
- TUI:
  - In some cases, the cursor in the Nvim TUI used to blink even without configuring 'guicursor' as mentioned in cursor-blinking. This was a bug that has now been fixed. If your cursor has stopped blinking, add the following (or similar, adapted to user preference) to your config file:

    ```
    set guicursor+=n-v-c:blinkon500-blinkoff500
    ```

NEW FEATURES

The following new features were added.

- API:
  - Passing 0 to nvim_get_chan_info() gets info about the current channel.
  - nvim_buf_set_extmark() supports inline virtual text.
  - nvim_win_text_height() computes the number of screen lines occupied by a range of text in a given window.
  - New RPC client type `msgpack-rpc` is added for nvim_set_client_info() to support fully MessagePack-RPC compliant clients.
  - Floating windows can now be hidden by setting `hide` in nvim_open_win() or nvim_win_set_config().
  - nvim_input_mouse() supports mouse buttons "x1" and "x2".
  - Added "force_crlf" option field in nvim_open_term().
  - Added nvim_tabpage_set_win() to set the current window of a tabpage.
  - Mapping APIs now support abbreviations when mode short-name has suffix "a".
  - Floating windows can now show footer with new `footer` and `footer_pos` config fields. Uses hl-FloatFooter by default.
  - extmarks can set a "url" highlight attribute, so the text region can become a clickable hyperlink (assuming UI support). The TUI renders URLs using the OSC 8 control sequence, enabling clickable text in supporting terminals.
  - nvim_open_win() and nvim_win_set_config() now support opening normal (split) windows, moving floating windows into split windows, and opening windows in non-current tabpages.
  - Flags added to nvim_buf_set_extmark():
    - "undo_restore": opt-out extmarks of precise undo tracking.
    - "invalidate": automatically hide or delete extmarks.
    - "virt_text_repeat_linebreak": repeat virtual text on wrapped lines.
  - Extmarks now fully support multi-line ranges, and a single extmark can be used to highlight a range of arbitrary length. The nvim_buf_set_extmark() API function already allowed you to define such ranges, but highlight regions were not rendered consistently for a range that covers more than one line break. This has now been fixed. Signs defined as part of a multi-line extmark also apply to every line in the range, not just the first. In addition, nvim_buf_get_extmarks() has gained an "overlap" option to return such ranges even if they started before the specified position.
- Defaults:
  - The `workspace/didChangeWatchedFiles` LSP client capability is now enabled by default on Mac and Windows. Disabled on Linux since there currently isn't a viable backend for watching files that scales well for large directories.
  - On Windows 'isfname' does not include ":". Drive letters are handled correctly without it. (Use gF for filepaths suffixed with ":line:col").
  - 'comments' includes "fb:•".
  - 'shortmess' includes the "C" flag.
  - 'grepprg' uses the -H and -I flags for grep by default, and defaults to using ripgrep if available.
  - "]d" and "[d" in Normal mode map to vim.diagnostic.goto_next() and vim.diagnostic.goto_prev(), respectively.]d-default [d-default
  - <C-W>d (and <C-W><C-D>) map to vim.diagnostic.open_float() CTRL-W_d-default
  - vim.lsp.start() sets the following default keymaps (assuming server support):
    - K in Normal mode maps to vim.lsp.buf.hover(), unless 'keywordprg' was customized before calling vim.lsp.start().
  - Automatic linting of treesitter query files (see ft-query-plugin). Can be disabled via:

    ```
    vim.g.query_lint_on = {}
    ```

  - Enabled treesitter highlighting for:
    - Treesitter query files
    - Vim help files
    - Lua files
- Editor:
  - Better cmdline completion for string option value. complete-set-option
    - Try it with `:set listchars=<Tab>`
  - By default, the swapfile "ATTENTION" E325 dialog is skipped if the swapfile is owned by a running Nvim process, instead of prompting. If you always want the swapfile dialog, delete the default SwapExists handler: `autocmd! nvim.swapfile`. default-autocmds
  - Navigating the jumplist with CTRL+O, CTRL+I behaves more intuitively when deleting buffers, and avoids "invalid buffer" cases. #25461
  - :fclose command.
  - v_Q-default and v_@-default repeat a register for each line of a linewise visual selection.
  - Clicking on a tabpage in the tabline with the middle mouse button closes it.
  - :checkhealth buffer can be opened in a split window using modifiers like :vertical, :horizontal and :botright.
- Events:
  - vim.on_key() callbacks receive a second argument for keys typed before mappings are applied.
- LSP:

    - LSP method names are available in `vim.lsp.protocol.Methods`.
    - Implemented LSP inlay hints: lsp-inlay_hint https://microsoft.github.io/language-server-protocol/specification/#textDocument_inlayHint
    - Implemented pull diagnostic textDocument/diagnostic: vim.lsp.diagnostic.on_diagnostic() https://microsoft.github.io/language-server-protocol/specification/#textDocument_diagnostic
    - Implemented LSP type hierarchy: vim.lsp.buf.typehierarchy() https://microsoft.github.io/language-server-protocol/specification/#textDocument_prepareTypeHierarchy
    - vim.lsp.status() consumes the last progress messages as a string.
    - LSP client now always saves and restores named buffer marks when applying text edits.
    - LSP client now supports the `positionEncoding` server capability. If a server responds with the `positionEncoding` capability in its initialization response, Nvim automatically sets the client's `offset_encoding` field.
    - Dynamic registration of LSP capabilities. An implication of this change is that checking a client's `server_capabilities` is no longer a sufficient indicator to see if a server supports a feature. Instead use `client.supports_method(<method>)`. It considers both the dynamic capabilities and static `server_capabilities`.
    - `anchor_bias` option to lsp-handlers aids in positioning of floating windows.
    - vim.lsp.util.locations_to_items() sets the `user_data` of each item to the original LSP `Location` or `LocationLink`.
    - Added support for connecting to servers using named pipes (Windows) or Unix domain sockets (Unix) via vim.lsp.rpc.connect().
    - Added support for `completionList.itemDefaults`, reducing overhead when computing completion items where properties often share the same value (e.g. `commitCharacters`). Note that this might affect plugins and language servers that don't support the feature, and in such cases the respective capability can be unset.
    - vim.lsp.start() accepts a "silent" option for suppressing messages if an LSP server failed to start.
    - vim.lsp.buf.definition(), vim.lsp.buf.declaration(), vim.lsp.buf.type_definition(), and vim.lsp.buf.implementation() now support the `loclist` field of vim.lsp.ListOpts.
- Lua:
    - :lua with a [range] executes that range as Lua code, in any buffer.
    - :source without arguments treats a buffer with 'filetype' of "lua" as Lua code regardless of its extension.
    - Vimscript function exists() supports checking v:lua functions.
    - vim.iter() is a generic interface for all iterable objects (tables, iterators).
    - vim.snippet provides a mode for expanding and navigating snippets.
    - vim.ringbuf() is a generic ring buffer (data structure).
    - vim.deepcopy() gained a `noref` argument to avoid hashing table values.
    - vim.keycode() translates keycodes in a string.
    - vim.system() runs commands / starts processes.
    - vim.lpeg and vim.re expose the bundled Lpeg expression grammar parser and its regex interface.
    - vim.base64.encode() and vim.base64.decode() encode and decode strings using Base64 encoding.
    - vim.text.hexencode() and vim.text.hexdecode() convert strings to and from byte representations.
    - vim.ui.open() opens URIs using the system default handler (macOS `open`, Windows `explorer`, Linux `xdg-open`, etc.)
    - vim.wo can now be double indexed for :setlocal behaviour. Currently only `0` for the buffer index is supported.
    - Improved messages for type errors in `vim.api.*` calls (including `opts` params).
    - Lua type annotations for:
        - `vim.*`
        - `vim.fn.*`
        - `vim.api.*`
        - `vim.v.*`
    - Functions that take a severity as an optional parameter (e.g. vim.diagnostic.get()) now also accept a list of severities vim.diagnostic.severity
    - vim.diagnostic.count() returns the number of diagnostics for a given buffer and/or namespace, by severity. This is a faster alternative to vim.diagnostic.get() when only the number of diagnostics is needed, but not the diagnostics themselves.
    - vim.diagnostic.is_enabled()
    - vim.version.le(), vim.version.ge()
    - vim.fs.root() finds project root directories from a list of "root markers".
    - vim.tbl_contains() now works for general tables and allows specifying a predicate function that is checked for each value. (Use vim.list_contains() for checking list-like tables (integer keys without gaps) for literal values.)
    - vim.region() can use a string accepted by getpos() as position.
- Options:
    - 'winfixbuf' keeps a window focused onto a specific buffer
    - 'smoothscroll' option to scroll by screen line rather than by text line when 'wrap' is set.
    - 'foldtext' now supports virtual text format. fold-foldtext
    - 'foldtext' can be set to an empty string to disable and render the line: as normal with regular highlighting and no line wrapping.
    - 'complete' option supports "f" flag for completing buffer names.
    - 'completeopt' option supports "popup" flag to show extra information in a floating window.
    - 'errorfile' (-q) accepts `-` as an alias for stdin.
- Performance:
    - 'diffopt' "linematch" scoring algorithm now favours larger and less groups https://github.com/neovim/neovim/pull/23611
    - Treesitter highlighting now parses injections incrementally during screen redraws only for the line range being rendered. This significantly improves performance in large files with many injections.
    - 'breakindent' performance is significantly improved for wrapped lines.
    - Cursor movement, insertion with [count] and screenpos() are now faster.
- Plugins:
    - Nvim now includes commenting support.
    - :Man supports the `:hide` modifier to open page in the current window.
    - :Man respects 'wrapmargin'
- Startup:
    - $NVIM_APPNAME can be set to a relative path instead of only a name.
    - --startuptime reports startup times for both processes (TUI + server) as separate sections.
- Terminal:
    - :terminal accepts some :command-modifiers (specifically :horizontal and those that affect splitting a window).
    - Terminal buffers emit a TermRequest autocommand event when the child process emits an OSC or DCS control sequence.

    - Terminal buffers respond to OSC background and foreground requests. default-autocmds
- Treesitter:
    - Bundled parser and queries (highlight, folds) for Markdown (used for LSP hover).
    - :InspectTree shows root nodes.
    - :InspectTree now supports folding.
    - :InspectTree shows node ranges in 0-based instead of 1-based indexing.
    - vim.treesitter.foldexpr() now recognizes folds captured using a quantified query pattern.
    - vim.treesitter.query.omnifunc() provides completion in treesitter query files (set by default).
    - vim.treesitter.query.edit() provides live editing of treesitter queries.
    - Query:iter_matches() now has the ability to set the maximum start depth for matches.
    - `@injection.language` now has smarter resolution and will fall back to language aliases (e.g., filetype or custom shorthands) registered via vim.treesitter.language.register() and/or attempt lower case variants of the text.
    - `@injection.filename` will try to match the node text via vim.filetype.match() and treat the result as a language name in the same way as `@injection.language`.
    - The `#set!` directive supports `injection.self` and `injection.parent` for injecting either the current node's language or the parent LanguageTree's language, respectively.
    - The `#set!` directive can set the "url" property of a node to have the node emit a hyperlink. Hyperlinks are UI specific: in the TUI, the OSC 8 control sequence is used.
    - Improved error messages for query parsing.
- TUI:
    - Builtin TUI can now recognize "super" (<D-) and "meta" (<T-) modifiers in a terminal emulator that supports tui-csiu.
    - The TermResponse event can be used with v:termresponse to read escape sequence responses from the host terminal.
    - A clipboard provider which uses OSC 52 to copy the selection to the system clipboard is now bundled by default and will be automatically enabled under certain conditions. clipboard-osc52
    - 'termsync' option asks the host terminal to buffer screen updates until the redraw cycle is complete. Requires support from the host terminal.
- UI:
    - Enhanced support for rendering multibyte characters using composing characters: the maximum limit was increased from 1+6 codepoints to 31 bytes, which is guaranteed to fit all chars from before but often more.
        - **NOTE:** the regexp engine still has a hard-coded limit of considering 6 composing chars only.

CHANGED FEATURES

These existing features changed their behavior.

- API:
    - nvim_buf_call() and nvim_win_call() now preserve any return value (NB: not multiple return values)
- Editor:
    - gx now uses vim.ui.open() and not netrw. To customize, you can redefine `vim.ui.open` or remap `gx`. To continue using netrw (deprecated):

      ```
      :packadd netrw
      :call netrw#BrowseX(expand(exists("g:netrw_gx") ? g:netrw_gx : '<cfile>'), netrw#CheckIfRemote())<CR>
      ```

- LSP:
    - LSP hover and signature help now use Treesitter for highlighting of Markdown content. Note that highlighting of code examples requires a matching parser and may be affected by custom queries.
    - LspRequest autocmd callbacks contain more information about the LSP request status update that occurred.
- Lua:
    - vim.wait() cannot be called in api-fast.
    - vim.diagnostic.config() now accepts virtual text relevant options to nvim_buf_set_extmark() (e.g. "virt_text_pos" and "hl_mode") in its "virtual_text" table, which gives users more control over how diagnostic virtual text is displayed.
    - vim.diagnostic.get() and vim.diagnostic.count() accept multiple namespaces rather than just a single namespace.
    - vim.diagnostic.enable() gained new parameters, and the old signature is deprecated.
    - vim.diagnostic.config() now accepts a function for the virtual_text.prefix option, which allows for rendering e.g., diagnostic severities differently.
- Options:
    - Attempting to set an invalid keycode option (e.g. `set t_foo=123`) no longer gives an error.
- Terminal:
    - Terminal buffers started with no arguments (and use 'shell') close automatically if the job exited without error, eliminating the (often unwanted) "[Process exited 0]" message. default-autocmds
- Treesitter:
    - Query:iter_matches(), vim.treesitter.query.add_predicate(), and vim.treesitter.query.add_directive() accept a new `all` option which ensures that all matching nodes are returned as a table. The default option `all=false` returns only a single node, breaking captures with quantifiers like `(comment)+ @comment`; it is only provided for backward compatibility and will be removed after Nvim 0.10.
    - vim.treesitter.query.add_predicate() and vim.treesitter.query.add_directive() now accept an options table rather than a boolean "force" argument. To force a predicate or directive to override an existing predicate or directive, use `{ force = true }`.

REMOVED FEATURES

These deprecated features were removed.

- Vimball support, including `:Vimuntar` command
- Support for legacy treesitter injection queries
- 'shortmess' flags:
    - shm-f. Always use "(3 of 5)", never "(file 3 of 5)".
    - shm-i. Always use "[noeol]".
    - shm-x. Always use "[dos]", "[unix]" and "[mac]".

- shm-n. Always use "[New]".

DEPRECATIONS

See deprecated-0.10.

Notable changes since Nvim 0.10

For changes in the previous release, see news-0.10.

BREAKING CHANGES

These changes may require adaptations in your config or plugins.

API

- `vim.rpcnotify(0)` and `rpcnotify(0)` broadcast to ALL channels. Previously they would "multicast" only to subscribed channels (controlled by `nvim_subscribe()`). Plugins and clients that want "multicast" behavior must now maintain their own list of channels.
  - In the future, vim.rpcnotify() may accept a list of channels, if there is demand for this use-case.
- "Dictionary" was renamed to "Dict" internally and in the RPC api-metadata. This is not expected to break clients because there are no known clients that actually use the `return_type` field or the parameter type names reported by --api-info or nvim_get_api_info().
- Renamed `nvim__id_dictionary` (unsupported/experimental API) to `nvim__id_dict`.

BUILD

- On Windows, only building with the UCRT runtime is supported.
- Translations are turned off by default. Enable by building Nvim with the CMake flag `ENABLE_TRANSLATIONS=ON`.

DIAGNOSTICS

- The "underline" diagnostics handler sorts diagnostics by severity when using the "severity_sort" option.
- Diagnostics are filtered by severity before being passed to a diagnostic handler diagnostic-handlers.
- The "virtual_text" handler is disabled by default. Enable with

```
vim.diagnostic.config({ virtual_text = true })
```

EDITOR

- The order in which signs are placed was changed. Higher priority signs will now appear left of lower priority signs.
- hl-CurSearch now behaves the same as Vim and no longer updates on every cursor movement.
- Moving in the buffer list using :bnext and similar commands behaves as documented and skips help buffers if run from a non-help buffer, otherwise it moves to another help buffer.
- Bells from a terminal buffer are now silent by default, unless 'belloff' option doesn't contain "term" or "all".
- Renamed autocmd group `nvim_swapfile` to `nvim.swapfile`. default-autocmds If you always want the swapfile dialog, delete the `nvim.swapfile` group: `autocmd! nvim.swapfile`.

EVENTS

- vim.ui_attach() callbacks for ui-messages `msg_show` events are executed in api-fast context.
- New/enhanced arguments in these existing UI events:
  - `cmdline_hide`: Includes `level` and `abort` arguments, `abort` argument indicating if the cmdline was aborted.
  - `cmdline_show`:
    - Prompts that were previously emitted as `msg_show` events, are now routed through `cmdline_show`.
    - `hl_id` argument to highlight the prompt text.
  - `msg_show`:
    - `history` argument indicating if the message was added to the history.
    - new message kinds: "bufwrite", "completion", "list_cmd", "lua_print", "search_cmd", "shell_out/err/ret", "undo", "verbose", wildlist".
- TermRequest and TermResponse event-data is now a table. The "sequence" field contains the received sequence. TermRequest also contains a "cursor" field indicating the cursor's position when the sequence was received.

HIGHLIGHTS

- TermCursorNC is removed and no longer supported. Unfocused terminals no longer have a cursor.

LSP

- vim.lsp.buf.references(), vim.lsp.buf.declaration(), vim.lsp.buf.definition(), vim.lsp.buf.type_definition(), vim.lsp.buf.implementation() and vim.lsp.buf.hover() now support merging the results of multiple clients but no longer trigger the global handlers from `vim.lsp.handlers`
- vim.lsp.buf.typehierarchy() now passes the correct params for each client request.
- vim.lsp.handlers.signature_help() is no longer used.
- vim.lsp.diagnostic.on_publish_diagnostics() and vim.lsp.diagnostic.on_diagnostic() no longer accept a config parameter and can no longer be configured with vim.lsp.with(). Instead use:

```
vim.diagnostic.config(config, vim.lsp.diagnostic.get_namespace(client_id))
```

- vim.lsp.util.make_position_params(), vim.lsp.util.make_range_params() and vim.lsp.util.make_given_range_params() now require the `position_encoding` parameter.
- vim.lsp.util.symbols_to_items() now requires the `position_encoding` parameter.

LUA

- API functions now consistently return an empty dictionary as vim.empty_dict(). Earlier, a lua-special-tbl was sometimes used.
- vim.json.encode() no longer escapes forward slashes "/" by default

OPTIONS

- The 'statuscolumn' `%l` item can now be used as a number column segment that changes according to related options. It takes care of alignment, 'number', 'relativenumber' and 'signcolumn' set to "number". The now redundant `%r` item is no longer treated specially for 'statuscolumn'.
- `:set {option}<` removes the local value for all global-local options instead of just string global-local options.
- `:setlocal {option}<` copies the global value to the local value for number and boolean global-local options instead of removing the local value.
- Setting hidden-options now gives an error. In particular, setting 'noshellslash' is now only allowed on Windows.

TREESITTER

- Query:iter_matches() correctly returns all matching nodes in a match instead of only the last node. This means that the returned table maps capture IDs to a list of nodes that need to be iterated over. For backwards compatibility, an option `all=false` (only return the last matching node) is provided that will be removed in a future release.
- vim.treesitter.language.get_filetypes() always includes the `{language}` argument in addition to explicitly registered filetypes.
- vim.treesitter.language.get_lang() falls back to the `{filetype}` argument if no languages are explicitly registered.
- vim.treesitter.language.add() returns `true` if a parser was loaded successfully and `nil,errmsg` otherwise instead of throwing an error.
- vim.treesitter.get_parser() and vim.treesitter.start() no longer parse the tree before returning. Scripts must call LanguageTree:parse() explicitly.

```
local p = vim.treesitter.get_parser(0, 'c')
p:parse()
```

- vim.treesitter.get_parser() expects its buffer to be loaded.

TUI

- OSC 52 is used as a fallback clipboard provider when no other clipboard-tool is found, even when not using SSH clipboard-osc52. To disable OSC 52 queries, set the "osc52" key of g:termfeatures to false.

VIMSCRIPT

- v:msgpack_types has the type "binary" removed. msgpackparse() no longer treats BIN, STR and FIXSTR as separate types. Any of these is returned as a string if possible, or a blob if the value contained embedded NUL:s.

NEW FEATURES

The following new features were added.

API

- Improved API "meta" docstrings and :help documentation.
- nvim__ns_set() can set properties for a namespace
- nvim_echo() `err` field to print error messages and chunks accepts highlight group IDs.
- nvim_open_win() supports a `mouse` field that allows configuring mouse interaction with the window separately from `focusable` field.
- nvim_open_win() `relative` field can be set to "laststatus" and "tabline".
- Additions to nvim_buf_set_extmark():
  - `conceal_lines` field to conceal an entire line.
  - `hl_group` field can be an array of layered groups.
  - `virt_text_pos` field accepts value `eol_right_align` to allow for right aligned text that truncates before covering up buffer text.
  - `virt_lines_overflow` field accepts value `scroll` to enable horizontal scrolling for virtual lines with 'nowrap'.

DEFAULTS

- Highlighting:
  - Improved styling of :checkhealth and :help buffers.
- Mappings:
  - grn in Normal mode maps to vim.lsp.buf.rename()
  - grr in Normal mode maps to vim.lsp.buf.references()
  - gri in Normal mode maps to vim.lsp.buf.implementation()
  - gO in Normal mode maps to vim.lsp.buf.document_symbol()
  - gra in Normal and Visual mode maps to vim.lsp.buf.code_action()
  - CTRL-S in Insert and Select mode maps to vim.lsp.buf.signature_help()
  - Mouse popup-menu includes an "Open in web browser" item when you right-click on a URL.
  - Mouse popup-menu includes a "Go to definition" item when LSP is active in the buffer.
  - Mouse popup-menu includes "Show Diagnostics", "Show All Diagnostics" and "Configure Diagnostics" items when there are diagnostics in the buffer.
  -]d-default and [d-default accept a count.
  - [D-default and]D-default jump to the first and last diagnostic in the current buffer, respectively.
  - Mappings inspired by Tim Pope's vim-unimpaired:
    - [q,]q, [Q,]Q, [_CTRL-Q,]_CTRL-Q navigate through the quickfix list
    - [l,]l, [L,]L, [_CTRL-L,]_CTRL-L navigate through the location-list
    - [t,]t, [T,]T, [_CTRL-T,]_CTRL-T navigate through the tag-matchlist
    - [a,]a, [A,]A navigate through the argument-list
    - [b,]b, [B,]B navigate through the buffer-list
    - [<Space>,]<Space> add an empty line above and below the cursor

    - [[and]] in Normal mode jump between shell prompts for shells which emit OSC 133 sequences ("shell integration" or "semantic prompts").
- Options:
    - 'diffopt' default includes "linematch:40".
    - 'number', 'relativenumber', 'signcolumn', and 'foldcolumn' are disabled in terminal buffers. terminal-config shows how to change these defaults.
    - Lua ftplugin sets 'omnifunc' to "v:lua.vim.lua_omnifunc".
    - Lua ftplugin sets 'foldexpr' to "v:lua.vim.treesitter.foldexpr()".
- Snippet:
    - <Tab> in Insert and Select mode maps to `vim.snippet.jump({ direction = 1 })` when a snippet is active and jumpable forwards.
    - <S-Tab> in Insert and Select mode maps to `vim.snippet.jump({ direction = -1 })` when a snippet is active and jumpable backwards.

DIAGNOSTICS

- vim.diagnostic.config() accepts a "jump" table to specify defaults for vim.diagnostic.jump().
- A "virtual_lines" diagnostic handler was added to render diagnostics using virtual lines below the respective code.
- The "virtual_text" diagnostic handler accepts a `current_line` option to only show virtual text at the cursor's line.

EDITOR

- Improved paste handling for redo (dot-repeat) and macros (recording):
    - Redoing a large paste is significantly faster and ignores 'autoindent'.
    - Replaying a macro with @ also replays pasted text.
- On Windows, filename arguments on the command-line prefixed with "~\" or "~/" are now expanded to the user's profile directory, not a relative path to a literal "~" directory.
- hl-ComplMatchIns shows matched text of the currently inserted completion.
- hl-PmenuMatch and hl-PmenuMatchSel show matched text in completion popup.
- gO now works in `help`, `checkhealth`, and `markdown` buffers.
- Jump between sections in `help` and `checkhealth` buffers with `[[` and `]]`.

EVENTS

- CompleteDone now sets the `reason` key in `v:event` which specifies the reason for completion being done.
- vim.on_key() callbacks can consume the key by returning an empty string.

LSP

- Improved rendering of LSP hover docs. K-lsp-default
- vim.lsp.completion.enable() gained the `convert` callback which enables customizing the transformation of an LSP CompletionItem to complete-items.
- vim.lsp.diagnostic.from() can be used to convert a list of vim.Diagnostic objects into their LSP diagnostic representation.
- `:checkhealth vim.lsp` displays the server version (if available).
- Completion side effects (including snippet expansion, execution of commands and application of additional text edits) is now built-in.
- vim.lsp.util.locations_to_items() and vim.lsp.util.symbols_to_items() now sets `end_col` and `end_lnum` fields.
- vim.lsp.buf.format() now supports passing a list of ranges via the `range` parameter (this requires support for the `textDocument/rangesFormatting` request).
- vim.lsp.buf.code_action() actions show client name when there are multiple clients.
- vim.lsp.buf.signature_help() can now cycle through different signatures using <C-s> and also support multiple clients.
- The client now supports `'utf-8'` and `'utf-32'` position encodings.
- vim.lsp.buf.hover() now highlights hover ranges using the hl-LspReferenceTarget highlight group.
- Functions in vim.lsp.Client can now be called as methods.
- Implemented LSP folding: vim.lsp.foldexpr() https://microsoft.github.io/language-server-protocol/specification/#textDocument_foldingRange
- vim.lsp.config() has been added to define default configurations for servers. In addition, configurations can be specified in `lsp/<name>.lua`.
- vim.lsp.enable() has been added to enable servers.
- vim.lsp.buf.code_action() resolves the `command` property during the `codeAction/resolve` request.
- The `textDocument/completion` request now includes the completion context in its parameters.

LUA

- Command-line completions for: `vim.g`, `vim.t`, `vim.w`, `vim.b`, `vim.v`, `vim.o`, `vim.wo`, `vim.bo`, `vim.opt`, `vim.opt_local`, `vim.opt_global`, `vim.env` and `vim.fn`.
- Documentation for lua-bit.
- gf in Lua buffers can go to module in same repo, runtime-search-path and package.path.
- vim.fs.rm() can delete files and directories.
- vim.validate() now has a new signature which uses less tables, is more performant and easier to read.
- vim.str_byteindex() and vim.str_utfindex() gained overload signatures supporting two new parameters, `encoding` and `strict_indexing`.
- vim.json.encode() has an option to enable forward slash escaping
- vim.fs.abspath() converts paths to absolute paths.
- vim.fs.relpath() gets relative path compared to base path.
- vim.fs.dir() and vim.fs.find() can now follow symbolic links, the behavior can be turn on using the new `follow` option.
- vim.hl.range() now has a optional `timeout` field which allows for a timed highlight.
- vim.text.indent() indents/dedents text.

OPTIONS

- 'completeopt' flag "fuzzy" enables fuzzy-matching during ins-completion.

- 'completeopt' flag "preinsert" highlights text to be inserted.
- 'wildmode' flag "noselect" shows 'wildmenu' without selecting an entry.
- 'messagesopt' configures :messages and hit-enter prompt.
- 'tabclose' controls which tabpage to focus when closing a tabpage.
- 'eventignorewin' to persistently ignore events in a window.
- 'winborder' sets the default border for floating-windows.

PERFORMANCE

- Significantly reduced redraw time for long lines with treesitter highlighting.
- LSP diagnostics and inlay hints are de-duplicated (new requests cancel inflight requests). This greatly improves performance with slow LSP servers.
- 10x speedup for vim.treesitter.foldexpr() (when no parser exists for the buffer).
- Strong treesitter-query caching makes repeat vim.treesitter.query.get() and vim.treesitter.query.parse() calls significantly faster for large queries.
- Treesitter highlighting is now asynchronous. To force synchronous parsing, use `vim.g._ts_force_sync_parsing = true`.
- Treesitter folding is now calculated asynchronously.
- LanguageTree:parse() now only runs the injection query on the provided range (as long as the language does not have a combined injection), significantly improving treesitter-highlight performance.
- Treesitter injection query iteration is now asynchronous, making edits in large buffers with combined injections much quicker.
- 10x reduction in blocking time when attaching an LSP to a large buffer.

PLUGINS

- EditorConfig
  - spelling_language property is now supported.
- 'inccommand' incremental preview can run on 'nomodifiable' buffers and restores their 'modifiable' state
- Commenting
  - 'commentstring' values can now be specified in a Treesitter capture's `bo.commentstring` metadata field, providing finer grained support for languages like JSX.

STARTUP

- -es ("script mode") disables shada by default.
- Nvim will fail if the --listen or $NVIM_LISTEN_ADDRESS address is invalid, instead of silently skipping an invalid address.

TERMINAL

- The terminal now understands the OSC 52 escape sequence to write to the system clipboard (copy). Querying with OSC 52 (paste) is not supported.
- hl-StatusLineTerm and hl-StatusLineTermNC define highlights for the status line in terminal windows.
- The terminal buffer now supports reflow (wrapped lines adapt when the buffer is resized horizontally). **Note:** Lines that are not visible and kept in 'scrollback' are not reflown.
- The terminal now supports OSC 8 escape sequences and will display hyperlinks in supporting host terminals.
- The terminal now uses the actual cursor, rather than a "virtual" cursor. This means that escape codes sent by applications running in a terminal buffer can change the cursor shape and visibility. However, it also means that the TermCursorNC highlight group is no longer supported: an unfocused terminal window will have no cursor at all (so there is nothing to highlight).
- jobstart() gained the "term" flag.
- The terminal will send theme update notifications when 'background' is changed and DEC mode 2031 is enabled.
- The terminal has experimental support for the Kitty keyboard protocol (sometimes called "CSI u" key encoding). Only the "Disambiguate escape codes" mode is currently supported.
- The terminal emits a TermRequest autocommand event when the child process emits an APC control sequence.
- TermRequest has a "cursor" field in its event-data indicating the cursor position when the sequence was received.

TREESITTER

- LanguageTree:node_for_range() gets anonymous and named nodes for a range
- vim.treesitter.get_node() now takes an option `include_anonymous`, default false, which allows it to return anonymous nodes as well as named nodes.
- treesitter-directive-trim! can trim all whitespace (not just empty lines) from both sides of a node.
- vim.treesitter.get_captures_at_pos() now returns the `id` of each capture
- New TSNode:child_with_descendant(), which efficiently gets the node's child that contains a given node as descendant.
- LanguageTree:parse() optionally supports asynchronous invocation, which is activated by passing the `on_parse` callback parameter.
- vim.treesitter.query.set() can now inherit and/or extend runtime file queries in addition to overriding.
- LanguageTree:is_valid() now accepts a range parameter to narrow the scope of the validity check.
- :InspectTree now shows which nodes are missing.
- Bundled markdown highlight queries use `conceal_lines` metadata to conceal code block fence lines vertically.
- vim.treesitter.language.inspect() shows additional information, including parser version for ABI 15 parsers.
- TSQuery:disable_pattern() and TSQuery:disable_capture() to turn off a specific pattern or capture in a query.
- vim.treesitter.get_captures_at_pos() returns the `pattern_id` of the pattern used to match each capture.
- Query:iter_captures() now accepts an `opts` parameter, similar to Query:iter_matches().

TUI

- The builtin UI declares info nvim_set_client_info() on its channel. See startup-tui. To see the current UI info, try this:

```
:lua =vim.api.nvim_get_chan_info(vim.api.nvim_list_uis()[1].chan)
```

- log messages written by the builtin UI client (TUI, --remote-ui) are now prefixed with "ui" instead of "?".
- The TUI will re-query the terminal's background color when a theme update notification is received and Nvim will update 'background' accordingly.

UI

- :detach the current UI, let the Nvim server continue running as a background process. Works with the builtin TUI, and all GUIs.
- vim.ui.open() (by default bound to gx) accepts an `opt.cmd` parameter which controls the tool used to open the given path or URL. If you want to globally set this, you can override vim.ui.open using the same approach described at vim.paste().
- `vim.ui.open()` now supports [lemonade](https://github.com/lemonade-command/lemonade) as an option for opening urls/files. This is handy if you are in an ssh connection and use `lemonade`.
- The ins-completion-menu now supports cascading highlight styles. hl-PmenuSel and hl-PmenuMatch both inherit from hl-Pmenu, and hl-PmenuMatchSel inherits highlights from both hl-PmenuSel and hl-PmenuMatch.
- vim.diagnostic.setqflist() updates an existing quickfix list with the given title if found
- ui-messages content chunks now also contain the highlight group ID.
- :checkhealth can display in a floating window, controlled by the g:health variable.

VIMSCRIPT

- getchar() and getcharstr() have optional `{opts}` Dict argument to control: cursor behavior, return type, and whether to simplify the returned key.

CHANGED FEATURES

These existing features changed their behavior.

- 'scrollbind' now works properly with buffers that contain virtual lines.

Scrollbind works by aligning to a target top line of each window in a tab page. Previously this was done by calculating the difference between the old top line and the target top line, and scrolling by that amount. Now the top lines are calculated using screen line numbers which take virtual lines into account.

- The implementation of grapheme clusters (or combining chars mbyte-combining) was upgraded to closely follow extended grapheme clusters as defined by UAX#29 in the unicode standard. Noteworthily, this enables proper display of many more emoji characters than before, including those encoded with multiple emoji codepoints combined with ZWJ (zero width joiner) codepoints.

This also applies to :terminal output, where width of cells will be calculated using the upgraded implementation.

- Custom highlights in 'rulerformat', 'statuscolumn', 'statusline', 'tabline', 'winbar', and the sign/number column are stacked with their respective highlight groups, as opposed to hl-Normal. This is also reflected in the `highlights` from nvim_eval_statusline(), with a new `groups` field containing an array of stacked highlight groups.
- vim.on_key() callbacks won't be invoked recursively when a callback itself consumes input.
- "q" in man pages now uses CTRL-W_q instead of CTRL-W_c to close the current window, and it no longer throws E444 when there is only one window on the screen. Global variable `vim.g.pager` is removed.
- Default 'titlestring' is now implemented with 'statusline' "%" format items. This means the default, empty value is essentially an alias to: `%t%(\ %M%)%(\ \(%{expand(\"%:~:h\")}\)%)%a\ -\ Nvim`. This is only an implementation simplification, not a behavior change.

REMOVED FEATURES

These deprecated features were removed.

- option `severity_limit` for `vim.lsp.diagnostic` (use `min=severity` instead vim.diagnostic.severity).

DEPRECATIONS

See deprecated-0.11.

Notable changes since Nvim 0.11

For changes in the previous release, see news-0.11.

BREAKING CHANGES

These changes may require adaptations in your config or plugins.

API

- nvim_get_commands() returns `complete` as a Lua function, if it was defined as such.

DIAGNOSTICS

- diagnostic-signs can no longer be configured with :sign-define or sign_define() (deprecated in Nvim 0.10 deprecated-0.10).
- Removed vim.diagnostic.disable() and vim.diagnostic.is_disabled() (deprecated in Nvim 0.10 deprecated-0.10).
- The legacy signature of vim.diagnostic.enable() (deprecated in Nvim 0.10 deprecated-0.10) is no longer supported.

EDITOR

- i_CTRL-R inserts named/clipboard registers (A-Z,a-z,0-9+) literally, like pasting instead of like user input. Improves performance, avoids broken formatting. To get the old behavior you can use `<C-R>=@x`.
- Parsing of URI-like buffer names more closely follows RFC3986, so for example "svn+ssh:", "ed2k:", and "iris.xpc:" are recognized as URI schemes.
- Security: Nvim on Windows no longer searches the current directory for external command executables; prefix a relative or absolute path if you want the old behavior $NoDefaultCurrentDirectoryInExePath.

EVENTS

- ui-messages no longer emits the `msg_show.return_prompt`, and `msg_history_clear` events. The `msg_clear` event was repurposed and is now emitted after the screen is cleared. These events arbitrarily assumed a message UI that mimics the legacy message grid. Benefit: reduced UI event traffic and more flexibility for UIs. The `msg_history_show` event has an additional "prev_cmd" argument.

LSP

- JSON "null" values in LSP messages are represented as vim.NIL instead of `nil`. Missing fields (as opposed to JSON "null") are still represented as `nil`.
- The function set with vim.lsp.log.set_format_func() is now given all arguments corresponding to a log entry instead of the individual arguments.
- Renamed `vim.lsp.semantic_tokens start()/stop()` to `enable()`.
- vim.lsp.util.convert_signature_help_to_markdown_lines() activeParameter handling updated:
  - Values < 0 are now treated as `nil` instead of 0.
  - Values outside the range of `signatures[activeSignature].parameters` are now treated as `nil` instead of `#signatures[activeSignature].parameters`

LUA

- Renamed `vim.diff` to `vim.text.diff`.

OPTIONS

- 'shelltemp' defaults to "false".

PLUGINS

- Removed "shellmenu" plugin, an old menu-based quasi-snippet plugin for shell scripts.
- package-tohtml is now an "opt-in" plugin. Use :packadd to activate it:

```
:packadd nvim.tohtml
```

TREESITTER

- ft-query-plugin no longer enables vim.treesitter.query.lint() by default.
- treesitter-directive-offset! can now be applied to quantified captures. It no longer sets `metadata[capture_id].range`; it instead sets `metadata[capture_id].offset`. The offset will be applied in vim.treesitter.get_range(), which should be preferred over reading metadata directly for retrieving node ranges.
- The "all" option to Query:iter_matches(), which was introduced in Nvim 0.11 to aid in transitioning to the new behavior, has been removed.

NEW FEATURES

The following new features were added.

API

- api-contract allows existing functions to change return-type from `void => non-void`.
- nvim_win_text_height() can limit the lines checked when a certain `max_height` is reached, and returns the `end_row` and `end_vcol` for which `max_height` or the calculated height is reached.
- vim.secure.read() now returns `true` for trusted directories. Previously it would return `nil`, thus impossible to tell if the directory was actually trusted.

- nvim_ui_send() writes arbitrary data to a UI's stdout. Use this to write escape sequences to the terminal when Nvim is running in the TUI.
- nvim_echo() can set the ui-messages kind with which to emit the message.
- nvim_echo() can create Progress messages
- nvim_get_commands() returns `preview` and `callback` as Lua functions if they were so specified in `nvim_create_user_command()`.
- nvim_open_tabpage() can open a new tabpage.
- nvim_open_win() floating windows can show a 'statusline'. Plugins can use `style='minimal'` or `:setlocal statusline=` to hide the statusline.
- nvim_win_set_config() can move floating-windows to other tabpages.
- EXPERIMENTAL: nvim__exec_lua_fast() allows remote API clients to execute Lua while Nvim is blocking for input. api-fast
- Decoration provider gained an `on_range` callback (deprecates `on_line`).
- vim.secure.trust() accepts `path` for the `allow` action.
- nvim_get_chan_info() includes `exitcode` field for :terminal buffers.
- nvim_set_hl() supports updating specified attributes only.

BUILD

- EXPERIMENTAL: Zig-based build is available as an alternative to CMake. It is currently limited in functionality, and CMake remains the recommended option for the time being.
- Nvim can be built without Unibilium (terminfo implementation), in which case the user's terminfo database won't be loaded and only internal definitions for the most common terminals are used:

```
make distclean && make CMAKE_EXTRA_FLAGS="-DENABLE_UNIBILIUM=0" DEPS_CMAKE_FLAGS="-DUSE_BUNDLED_UNIBILIUM=0"
```

- On Windows, `tee.exe` is included with `nvim.exe` by default so that commands like :make, :grep work out of the box.

DEFAULTS

- 'diffopt' default value now includes "indent-heuristic" and "inline:char".
- 'statusline' default is exposed as a statusline expression (previously it was implemented as an internal C routine).
- Default 'statusline' shows:
  - vim.diagnostic.status()
  - vim.ui.progress_status()
  - :terminal exit code
- Project-local configuration ('exrc') is also loaded from parent directories. Unset 'exrc' to stop further search.
- Mappings:
  - grt in Normal mode maps to vim.lsp.buf.type_definition()
  - grx in Normal mode maps to vim.lsp.codelens.run()
- 'shada' default now excludes "/tmp/" and "/private/" paths to reduce clutter in :oldfiles.
- Enabled treesitter highlighting for Markdown files.

DIAGNOSTICS

- vim.diagnostic.setloclist() and vim.diagnostic.setqflist() now support a `format` function to modify (or filter) diagnostics before being set in the location/quickfix list.
- vim.diagnostic.get() now accepts an `enabled` filter to only return enabled or disabled diagnostics.
- vim.diagnostic.status() returns a status description of current buffer diagnostics.
- vim.diagnostic.fromqflist() accepts `opts.merge_lines` to merge multiline compiler messages.

EDITOR

- :iput works like :put but adjusts indent.
- :retab accepts new optional parameter -indentonly to only change leading whitespace in indented lines.
- :uniq deduplicates text in the current buffer.
- omnicompletion is available when editing help buffers. ft-help-omni
- :help! has DWIM ("Do What I Mean") behavior: it tries to guess the help tag at cursor. In help buffers, 'keywordprg' defaults to ":help!". For example, try "K" anywhere in this code:

```
local v = vim.version.parse(vim.system({'foo'}):wait().stdout)
```

- Setting "'0" in 'shada' prevents storing the jumplist in the shada file.
- 'shada' now correctly respects "/0" and "f0".
- prompt-buffer supports multiline input/paste, undo/redo, and o/O normal commands.
- 'wildchar' now enables completion in search contexts using /, ?, :g, :v and :vimgrep commands.
- Security: 'exrc' no longer shows "(a)llow". Instead you must "(v)iew", then run `:trust` (or `:trust [file]`, but that has a TOCTOU risk).
- gx in help buffers opens the online documentation for the tag at cursor.
- :source with a range in non-Lua files (e.g., vimdoc) now detects Lua codeblocks via treesitter and executes them as Lua instead of Vimscript. (Or use `:{range}lua` to skip the detection.)
- :wall with ++p auto-creates missing parent directories.

EVENTS

- In the ui-protocol, message kind `empty` is emitted for empty messages (e.g. `:echo ""`).
- CmdlineLeave sets v:char to the character that stops the Cmdline mode.
- CmdlineLeavePre triggered before preparing to leave the command line.
- New append, `id` and `trigger` parameter for ui-messages `msg_show` event.
- 'rulerformat' is emitted as `msg_ruler` when not part of the statusline.
- Creating or updating a progress message with nvim_echo() triggers a Progress event.
- MarkSet is triggered after a mark is set by the user (currently doesn't support implicit marks like '[or '<, ...).

- SessionLoadPre is triggered before loading a Session file.
- TabClosedPre is triggered before closing a tabpage.
- TermRequest event gained a `terminator` parameter.

HIGHLIGHTS

- hl-DiffTextAdd highlights added text within a changed line.
- hl-OkMsg hl-StderrMsg hl-StdoutMsg
- hl-SnippetTabstopActive highlights the currently active tabstop.
- hl-PmenuBorder hl-PmenuShadow hl-PmenuShadowThrough see 'pumborder'.
- nvim_set_hl() and nvim_get_hl() support the SGR attributes "dim", "blink", "conceal", and "overline".

LSP

- lsp-completion:
  - vim.lsp.completion.enable() gained a `cmp` option for custom ordering.
  - Supports colored symbol preview for color items.
  - Shows a preview ("completionItem/resolve") if 'completeopt' has "popup".
  - Support for `textDocument/inlineCompletion` lsp-inline_completion https://microsoft.github.io/language-server-protocol/specifications/lsp/3.18/specification/#textDocument_inlineCompletion See lsp-inline_completion for quickstart instructions.
- :lsp can be used to interactively manage LSP clients.
- vim.lsp.ClientConfig gained `workspace_required`.
- You can control the priority of vim.lsp.Config `root_markers`.
- LSP capabilities:
  - `textDocument/codeLens` lsp-codelens was reimplemented. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.18/specification/#textDocument_codeLens
  - `textDocument/colorPresentation` lsp-document_color https://microsoft.github.io/language-server-protocol/specification/#textDocument_colorPresentation
  - `textDocument/diagnostic`
  - `textDocument/documentColor` lsp-document_color https://microsoft.github.io/language-server-protocol/specification/#textDocument_documentColor
  - `textDocument/documentLink`: gx opens "documentLink" items at cursor.
  - `textDocument/inlineCompletion`
  - `textDocument/linkedEditingRange` lsp-linked_editing_range https://microsoft.github.io/language-server-protocol/specification/#textDocument_linkedEditingRange
  - `textDocument/onTypeFormatting` lsp-on_type_formatting https://microsoft.github.io/language-server-protocol/specification/#textDocument_onTypeFormatting
  - `textDocument/selectionRange` Incremental selection. v_an selects outwards, v_in selects inwards.
  - `textDocument/semanticTokens/range`
  - `workspace/codeLens/refresh` https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#codeLens_refresh
  - `workspace/diagnostic/refresh` https://microsoft.github.io/language-server-protocol/specification/#diagnostic_refresh
  - `workspace/diagnostic` vim.lsp.buf.workspace_diagnostics() https://microsoft.github.io/language-server-protocol/specifications/specification-current/#workspace_dagnostics
- Support for dynamic registration:
  - `textDocument/diagnostic`
- The `textDocument/diagnostic` request now includes the previous id in its parameters.
- vim.lsp.enable() start/stops clients as necessary and detaches non-applicable LSP clients.
- vim.lsp.is_enabled() checks if a LSP config is enabled (without "resolving" it).
- vim.lsp.get_configs() gets all LSP configs matching an optional filter.
- Support for multiline semantic tokens.
- Support for the `disabled` field on code actions.
- The function form of `cmd` in a vim.lsp.Config or vim.lsp.ClientConfig receives the resolved config as the second arg: `cmd(dispatchers, config)`.
- Support for annotated text edits.
- `:checkhealth vim.lsp` is now available to check which buffers the active LSP features are attached to.
- LSP `DiagnosticRelatedInformation` is now shown in vim.diagnostic.open_float(). It is read from the LSP diagnostic object stored in the `user_data` field.
- When inside the float created by vim.diagnostic.open_float() and the cursor is on a line with `DiagnosticRelatedInformation`, gf can be used to jump to the problematic location.
- Support for related documents in pull diagnostics: https://microsoft.github.io/language-server-protocol/specifications/specification-current/#relatedFullDocumentDiagnosticReport
- vim.lsp.buf.signature_help() supports "noActiveParameterSupport".
- The filter option of vim.lsp.buf.code_action() now receives the client ID as an argument.
- vim.lsp.ClientConfig `exit_timeout` decides the time waited before "stop" escalates to "force-stop" for vim.lsp.enable(), Client:stop(), and during Nvim shutdown.
  - `exit_timeout` graduated from "experimental" `flags.exit_timeout` to a top-level field. Defaults to `false`.
- Client:stop() accepts `force` as an integer, which is treated as the time to wait before before stop escalates to force-stop.
- vim.lsp.buf.rename() now highlights the symbol being renamed using the hl-LspReferenceTarget highlight group.
- Code lenses now display as virtual lines

LUA

- vim.net.request() can fetch/download HTTP content.
- vim.wait() returns the callback results.
- Lua type annotations for `vim.uv`.
- vim.hl.range() now allows multiple timed highlights.

- vim.tbl_extend(), vim.tbl_deep_extend() `behavior` argument can be a function.
- vim.fs.root() can define "equal priority" via nested lists.
- vim.fs.ext() returns the last extension of a file.
- vim.version.range() can output human-readable string via tostring().
- vim.version.intersect() computes intersection of two version ranges.
- Iter:take() and Iter:skip() now optionally accept predicates.
- Iter:peek() now works for all iterator types, not just list-iterator.
- vim.list.unique() and Iter:unique() to deduplicate lists and iterators, respectively.
- vim.list.bisect() performs binary search.
- vim.json.encode() `indent` option performs pretty-formatting.
- vim.json.encode() `sort_keys` option sorts by key.
- vim.json.decode() `skip_comments` option allows comments in JSON data.
- EXPERIMENTAL: vim.pos, vim.range provide Position/Range abstraction.

OPTIONS

- 'autocomplete' enables ins-autocompletion.
- 'autowriteall' writes all buffers upon receiving `SIGHUP`, `SIGQUIT` or `SIGTSTP`.
- 'chistory' and 'lhistory' set size of the quickfix-stack.
- 'complete' new flags:
  - "F{func}" complete using given function
  - "F" complete using 'completefunc'
  - "o" complete using 'omnifunc'
- 'complete' allows limiting matches for sources using "{flag}^<limit>".
- 'completeopt' flag "nearest" sorts completion results by distance to cursor.
- 'diffanchors' specifies addresses to anchor a diff.
- 'diffopt' `inline:` configures diff highlighting for changes within a line.
- 'diffopt' with `inline:word` now automatically merges adjacent diff blocks separated by gaps and punctuation by 5 bytes to improve readability.
- 'fillchars' has new flag "foldinner".
- 'fsync' and 'grepformat' are now global-local options.
- 'listchars' has new flag "leadtab".
- 'jumpoptions' flag "view" now applies when popping the tagstack.
- 'maxsearchcount' sets maximum value for searchcount() and defaults to 999.
- 'pummaxwidth' sets maximum width for the completion popup menu.
- 'winborder' "bold" style, custom border style.
- 'busy' sets a buffer "busy" status. Indicated in the default statusline.
- 'pumborder' adds a border to the popup menu.
- g:clipboard accepts a string name to force any builtin clipboard tool.
- g:clipboard autodetection selects tmux only when running inside tmux.
- 'statusline' allows "stacking" highlight groups (groups inherit from previous highlight attributes)
- 'messagesopt' "progress:c" flag controls whether progress messages are shown in the cmdline message area.
- Improved :set+=, :set^= and :set-= handling of comma-separated "key:value" pairs (e.g. 'listchars', 'fillchars', 'diffopt').

PERFORMANCE

- vim.glob.to_lpeg() uses a new LPeg-based implementation (Peglob) that provides ~50% speedup for complex patterns. The implementation restores support for nested braces and follows LSP 3.17 specification with additional constraints for improved correctness and resistance to backtracking edge cases.
- i_CTRL-R inserts named/clipboard registers literally, 10x speedup.
- LSP `textDocument/semanticTokens/range` is supported, which requests tokens for the viewport (visible screen) only.
- :packadd doesn't invalidate the cached Lua package path. Instead the cache gets updated in place. This might make a big startuptime difference for certain init.lua patterns where multiple :packadd or vim.pack.add() calls are interspersed with other code.

PLUGINS

- Built-in plugin manager: vim.pack
- :DiffTool compares directories (and files).
- :Undotree lets you visually navigate the undo-tree.
- Customize :checkhealth by handling a `FileType checkhealth` event. health-usage
- Simplify Python provider setup to a single step: `uv tool install pynvim` Nvim will detect the plugin's location without user configuration, even if unrelated Python virtual environments are activated. provider-python
- :checkhealth now checks for an available vim.ui.open() handler.
- spellfile.vim : Replaced by package-spellfile.

STARTUP

- v:argf provides file arguments given at startup.
- Nvim shows a warning if the log file path is inaccessible.

TERMINAL

- nvim_open_term() can be called with a non-empty buffer. The buffer contents are piped to the PTY and displayed as terminal output.
- CSI 3 J (the sequence to clear terminal scrollback) is now supported.
- DEC private mode 2026 (synchronized output) is now supported. Applications running in :terminal can batch screen updates to avoid tearing.
- A suspended PTY process is now indicated by "[Process suspended]" at the bottom-left of the buffer and can be resumed by pressing a key.
- On terminal exit, "[Process exited]" is shown as virtual text (instead of modifying buffer contents), and exit code is shown in statusline.

- You can disable "[Process exited]" entirely, see terminal-config.

TREESITTER

- Query:iter_captures() supports specifying starting and ending columns.
- :EditQuery command gained tab-completion, works with injected languages.
- LanguageTree:parse() now accepts a list of ranges.
- v_an v_in v_]n v_[n incremental selection of treesitter nodes.

TUI

- TermResponse now supports DA1 and APC query responses.
- Native progress bars are displayed for Progress events using the OSC 9;4 sequence.
- The TUI now renders the SGR dim (faint), blink, conceal, and overline attributes.

UI

- ui2 is a redesign of the core messages and commandline UI, which will replace the legacy message grid in the TUI.
  - Avoids "Press ENTER" interruptions.
  - Avoids delays from W10 "Changing a readonly file" and other warnings.
  - Highlights the cmdline as you type.
  - Provides the pager as a buffer + window.
  - Currently experimental. To enable it: `require('vim._core.ui2').enable()`
- :restart restarts Nvim and reattaches all UIs.
- :connect dynamically connects the current UI to the server at the given address.
- :checkhealth shows a summary in the header for every healthcheck.
- ui-multigrid provides composition information and absolute coordinates.
- Error messages are more concise:
  - "Error detected while processing:" changed to "Error in:".
  - "Error executing Lua:" changed to "Lua:".
- 'busy' status is shown in default statusline with symbol ◐
- Cursor shape indicates when it is behind an unfocused floating window.
- Improved LSP signature help rendering.
- Multigrid UIs can call nvim_input_mouse with grid 0 to let Nvim decide the grid.
- vim.ui.progress_status() returns a status description of currently running progress-messages.

VIMSCRIPT

- chdir() allows optionally specifying a scope argument.
- cmdcomplete_info() gets current cmdline completion info.
- getcompletiontype() gets command-line completion type for any string.
- has() gained two new system dependent feature flags, android and termux.
- prompt_getinput() gets current user-input in prompt-buffer.
- wildtrigger() triggers command-line expansion.
- v:vim_did_init is set after sourcing init.vim but before load-plugins.
- prompt_appendbuf() appends text to prompt-buffer.

CHANGED FEATURES

These existing features changed their behavior.

- 'pumblend' does not apply special attributes (bold, underline) from the background layer to the foreground layer.
- gv works in operator pending mode and does not abort.
- 'smartcase' applies to completion filtering.
- 'spellfile' location defaults to `stdpath("data")."/site/spell/"` instead of the first writable directory in 'runtimepath'.
- vim.version.range() doesn't exclude `to` if it is equal to `from`.
- $VIM and $VIMRUNTIME no longer check for Vim version-specific runtime directory `vim{number}` (e.g. `vim82`).
- 'scrollback' maximum value increased from 100000 to 1000000
- matchfuzzy() and matchfuzzypos() use an improved fuzzy matching algorithm (same as fzy).
- Windows: Paths like "\Windows" and "/Windows" are now considered to be absolute paths (to the current drive) and no longer relative.
- When 'shelltemp' is off, shell commands now use `pipe()` and not `socketpair()` for input and output. This matters mostly for Linux where some command lines using "/dev/stdin" and similar would break as these special files can be reopened when backed by pipes but not when backed by socket pairs.

REMOVED FEATURES

These deprecated features were removed.

- vim.treesitter.get_parser() instead of throwing an error `get_parser()` now always returns nil when it fails to create a parser.

DEPRECATIONS

See deprecated-0.12.

Notable changes since Nvim 0.8

BREAKING CHANGES

The following changes may require adaptations in user config or plugins.

- Cscope support is now removed (see cscope and nvim-removed):
  - Commands removed:
    - `:cscope`
    - `:lcscope`
    - `:scscope`
    - `:cstag`
  - Options removed:
    - `cscopepathcomp`
    - `cscopeprg`
    - `cscopequickfix`
    - `cscoperelative`
    - `cscopetag`
    - `cscopetagorder`
    - `cscopeverbose`
  - Eval functions removed:
    - `cscope_connection()`

Note: support for ctags remains with no plans to remove.

See https://github.com/neovim/neovim/pull/20545 for more information.

- `:hardcopy` is now removed (see hardcopy and nvim-removed):
  - Commands removed:
    - `:hardcopy`
  - Options removed:
    - `printdevice`
    - `printencoding`
    - `printexpr`
    - `printfont`
    - `printheader`
    - `printmbcharset`
- 'paste' option is now deprecated and 'pastetoggle' is removed. paste works automatically in GUI and terminal (TUI) Nvim. Just Paste It.™
- Changes to vim.treesitter.get_node_text():
  - It now returns `string`, as opposed to `string|string[]|nil`.
  - The `concat` option has been removed as it was not consistently applied.
  - Invalid ranges now cause an error instead of returning `nil`.
- `help` treesitter parser was renamed to `vimdoc`. The only user-visible change is that language-specific highlight groups need to be renamed from `@foo.help` to `@foo.vimdoc`.
- The default value of 'commentstring' is now empty instead of "/*%s*/".
- libiconv and intl are now required build dependencies.

NEW FEATURES

The following new APIs or features were added.

- Treesitter syntax highlighting for `help` files now supports highlighted code examples. To enable, create a `.config/nvim/ftplugin/help.lua` with the contents

```
vim.treesitter.start()
```

  - **Note:** Highlighted code examples are only available in the Nvim manual, not in help files taken from Vim. The treesitter `vimdoc` parser is also work in progress and not guaranteed to correctly highlight every help file in the wild.
- Added support for semantic token highlighting to the LSP client. This functionality is enabled by default when a client that supports this feature is attached to a buffer. Opt-out can be performed by deleting the `semanticTokensProvider` from the LSP client's `{server_capabilities}` in the `LspAttach` callback. See lsp-semantic-highlight for more information.
- vim.inspect_pos(), vim.show_pos() and :Inspect allow a user to get or show items at a given buffer position. Currently this includes treesitter captures, LSP semantic tokens, syntax groups and extmarks.
- vim.treesitter.inspect_tree() and :InspectTree opens a split window showing a text representation of the nodes in a language tree for the current buffer.
- 'statuscolumn' option to customize the area to the side of a window, normally containing the fold, sign and number columns. This new option follows the 'statusline' syntax and can be used to transform the line numbers, create mouse click callbacks for signs, introduce a custom margin or separator etc.
- vim.secure.trust(), :trust allows the user to manage files in trust database. vim.secure.read() reads a file and prompts the user if it should be trusted and, if so, returns the file's contents. Used by 'exrc'
- EditorConfig support is now builtin. This is enabled by default and happens automatically. To disable it, add the following to your config:

```
vim.g.editorconfig = false
```

- A new environment variable named NVIM_APPNAME enables configuring the directories where Nvim should find its configuration and state files. See `:help $NVIM_APPNAME` .
- Added support for running Lua scripts from shell using -l.

```
nvim -l foo.lua --arg1 --arg2
```

Also works with stdin:

```
echo "print(42)" | nvim -l -
```

- Added an omnifunc implementation for Lua, vim.lua_omnifunc()
- Added a new experimental vim.loader that byte-compiles and caches Lua files. To enable the new loader, add the following at the top of your init.lua:

```
vim.loader.enable()
```

- Added vim.version for parsing and comparing version strings conforming to the semver specification.
- When using Nvim inside tmux 3.2 or later, the default clipboard provider will now copy to the system clipboard. provider-clipboard
- 'showcmdloc' option to display the 'showcmd' information in the status line or tab line. A new %S statusline item is available to place the 'showcmd' text in a custom 'statusline'. Useful for when 'cmdheight' is set to 0.
- 'splitkeep' option to control the scroll behavior of horizontal splits.
- 'wildoptions' flag "fuzzy" enables fuzzy-matching during cmdline-completion.
- 'diffopt' now includes a `linematch` option to enable a second-stage diff on individual hunks to provide much more accurate diffs. This option is also available to vim.text.diff()

See https://github.com/neovim/neovim/pull/14537.

- --remote-ui option was added to connect to a remote instance and display in it in a TUI in the local terminal. This can be used run a headless nvim instance in the background and display its UI on demand, which previously only was possible using an external UI implementation.
- Added a vim.lsp.codelens.clear() function to clear codelenses.
- Added support for the `willSave` and `willSaveWaitUntil` capabilities to the LSP client. `willSaveWaitUntil` allows a server to modify a document before it gets saved. Example use-cases by language servers include removing unused imports, or formatting the file.
- Added preliminary support for the `workspace/didChangeWatchedFiles` capability to the LSP client to notify servers of file changes on disk. The feature is disabled by default and can be enabled by setting the `workspace.didChangeWatchedFiles.dynamicRegistration=true` capability.
- vim.diagnostic now supports LSP DiagnosticsTag. See: https://microsoft.github.io/language-server-protocol/specification/#diagnosticTag
- vim.diagnostic.is_disabled() checks if diagnostics are disabled in a given buffer or namespace.
- Treesitter captures can now be transformed by directives. This will allow more complicated dynamic language injections.
- vim.treesitter.get_node_text() now accepts a `metadata` option for writing custom directives using vim.treesitter.query.add_directive().
- vim.treesitter.language.add() replaces `vim.treesitter.language.require_language`.
- vim.treesitter.foldexpr() can be used for 'foldexpr' to use treesitter for folding.
- Expanded the TSNode API with:
  - TSNode:tree()
  - TSNode:has_changes()
  - TSNode:extra()
  - TSNode:equal()

Additionally TSNode:range() now takes an optional `{include_bytes}` argument.

- Treesitter injection queries now use the format described at https://tree-sitter.github.io/tree-sitter/3-syntax-highlighting.html#language-injection . Support for the previous format will be removed in a future release.
- Added nvim_get_hl() for getting highlight group definitions in a format compatible with nvim_set_hl().
- vim.filetype.get_option() to get the default option value for a specific filetype. This is a wrapper around nvim_get_option_value() with caching.
- `require'bit'` is now always available lua-bit

CHANGED FEATURES

The following changes to existing APIs or features add new behavior.

- 'exrc' now supports `.nvim.lua` file.
- 'exrc' is no longer marked deprecated.
- The TUI is changed to run in a separate process (previously, a separate thread was used). This is not supposed to be a visible change to the user, but might be the cause of subtle changes of behavior and bugs.

Previously, the TUI could be disabled as a build time feature (+tui/-tui), resulting in a nvim binary which only could be run headless or embedded in an external process. As of this version, TUI is always available.

- Vim's `has('gui_running')` is now supported as a way for plugins to check if a GUI (not the TUI) is attached to Nvim. has()
- msgsep is now always enabled even if 'display' doesn't contain the "msgsep" flag. It is no longer possible to scroll the whole screen when showing messages longer than 'cmdheight'.
- API calls now show more information about where an exception happened.
- The `win_viewport` UI event now contains information about virtual lines, meaning that smooth scrolling can now be implemented more consistently.
- The `:= {expr}` syntax can be used to evaluate a Lua expression, as a shorter form of `:lua ={expr}`. `:=` and `:[range]=` without argument are unchanged. However `:=#` and similar variants using ex-flags are no longer supported.
- Unsaved changes are now preserved rather than discarded when channel-stdio is closed.
- nvim_open_win() now accepts a relative `mouse` option to open a floating win relative to the mouse. Note that the mouse doesn't update frequently without setting `vim.o.mousemoveevent = true`
- nvim_eval_statusline() supports evaluating the 'statuscolumn' through a new `opts` field: `use_statuscol_lnum`.
- nvim_buf_get_extmarks() now accepts a `-1` `ns_id` to request extmarks from all namespaces and adds the namespace id to the details array. Other missing properties have been added to the details array and marks can be filtered by type.
- vim.diagnostic.open_float() (and therefore vim.diagnostic.config()) now accepts a `suffix` option which, by default, renders LSP error codes. Similarly, the `virtual_text` configuration in vim.diagnostic.config() now has a `suffix` option which does nothing by default.
- vim.fs.dir() now has a `opts` argument with a depth field to allow recursively searching a directory tree.
- vim.gsplit() supports all features of vim.split().

- :highlight now supports an additional attribute "altfont".
- :Man manpage viewer supports manpage names containing spaces.
- nvim_select_popupmenu_item() now supports cmdline-completion popup menu.
- nvim_list_uis() reports all ui-option fields.
- nvim_get_option_value() now has a `filetype` option so it can return the default option for a specific filetype.
- build: Several improvements were made to make the code generation scripts more deterministic, and a `LUA_GEN_PRG` build parameter has been introduced to allow for a workaround for some remaining reproducibility problems.

REMOVED FEATURES

The following deprecated functions or APIs were removed.

- `filetype.vim` is removed in favor of vim.filetype (Note that filetype logic and tests still align with Vim, so additions or changes need to be contributed there first.) See https://github.com/neovim/neovim/pull/20674.
- 'hkmap', 'hkmapp' and 'aleph' options were removed. Use 'keymap' option instead.
- LanguageTree:parse() no longer returns changed regions. Please use the `on_changedtree` callbacks instead.
- `vim.highlight.create()`, `vim.highlight.link()` were removed, use nvim_set_hl() instead.
- `require'health'` was removed. Use vim.health instead.

DEPRECATIONS

See deprecated-0.9.

Notable changes since Nvim 0.12

For changes in the previous release, see news-0.12.

BREAKING CHANGES IN HEAD OR EXPERIMENTAL news-breaking-dev

====== Remove this section before release. ======

The following changes to UNRELEASED features were made during the development cycle (Nvim HEAD, the "master" branch).

EVENTS

- todo

EXPERIMENTS

- todo

LSP

- todo

LUA

- vim.pos, vim.range always require the `buf` parameter.
- range.cursor() and range.to_cursor() are removed. Use range.mark() and range.to_mark() instead.
- pos.to_cursor() returns a `(row, col)` tuple instead of returning them as separate values.
- Renamed current_level param of vim.log.new().
- `vim.opt.{option}:get()` has improved structure for dict-like options. Some shapes changed:
  - "key:value" list options (e.g. 'diffopt', 'mousescroll') return a map `{ key = value }` instead of an array of `"key:value"` strings.
  - The `,,` literal-comma convention (e.g. 'isfname') is not reconstructed in the structured view; the raw string value is unchanged.

DIAGNOSTICS

- todo

OPTIONS

- When ui2 is enabled, since the ruler is no longer a left-aligned box of fixed width, there is no longer an inconsistent special handling of item groups in 'rulerformat' to configure that fixed width. An old-style left- aligned fixed-width ruler can now be achieved by using item groups exactly as they are documented, e.g. `%-20.20(...%)`.

TREESITTER

- todo

UI

- Healthchecks for vim.ui.img were moved to `:checkhealth vim.health`.
- ui2 messages pager no longer uses <CR> by default; use g< or set 'messagesopt' "pager", e.g. `:set messagesopt+=pager:<CR>`.
- ui2 `cfg.pager_char` and `cfg.msg.msg.timeout` were replaced by the 'messagesopt' items "pager" and "timeout".

VIMSCRIPT

- todo

BREAKING CHANGES news-breaking

These changes may require adaptations in your config or plugins.

API

- API results containing a Lua function reference (e.g. a "callback" reported by nvim_get_keymap() or nvim_get_autocmds(), or a "callback" option value) are serialized over RPC as a human-readable "<Lua …>" hint string, instead of nil.
- nvim_create_autocmd(), nvim_exec_autocmds() and nvim_clear_autocmds() no longer treat an empty non-nil pattern as nil.
- nvim_clear_autocmds() no longer treats an empty array event as nil.
- vim.o, vim.opt, and nvim_set_option_value() expand ~ and environment variables (expand-env).

DIAGNOSTICS

- `vim.diagnostic.Opts.Status.format` no longer accepts the table mapping from severity to text. Use `vim.diagnostic.Opts.Signs.text` instead.
- `current_line` property of vim.diagnostic.Opts.VirtualLines and vim.diagnostic.Opts.VirtualText is now applied on CursorHold event. Make sure your 'updatetime' is reasonable.

EDITOR

- Interactive Ex-mode (:exmode, `nvim -e`) is a persistent cmdwin. This is much more flexible and makes "Ex mode" more useful as a kind of "cmdline REPL", with the tradeoff of dropping some POSIX Ex-mode quirks.

    - mode() returns "n" or "i"; "cv" indicates -es (script mode) only.
    - Ex mode is entered with "1q:" or :exmode, not "gQ".
    - ":global/pat/visual" no longer switches to Normal mode per matching line.
- Non-interactive Ex-mode (-es, "script mode") executes stdin as Ex commands directly. It is a "script host", not "POSIX Ex mode".
    - Output ends with a final newline.
    - :visual is :edit: it does not switch to reading stdin as Normal-mode commands, and ":global/pat/visual" does not enter Normal mode per matching line.
    - Commands that wait for typed input (input(), getchar(), ...) stop Nvim at end-of-input instead of consuming the next stdin line.
    - An empty line is a no-op instead of advancing the cursor (or failing with E501 at end-of-file).
- On Windows, the trust db now stores paths with "/" slashes. This means the trust store will be reset (one time).
- `stdpath("log")` moved to `stdpath("state")/logs`.
- An operator on a target that creates a Visual selection ("cgn") does not change '< '>. gv
- cmdwin (q:, q/, q?, c_CTRL-F):
    - Only supports one "cmdline level", not "infinite recursion".
    - Expr-register (i_CTRL-R_=, c_CTRL-R_=) no longer supports opening cmdwin via c_CTRL-F.
- Behavior of :restart changed. Use "!" (:restart!) to get the old behavior.
    - ZR now performs :restart. Add a count to change the behavior.
- ins-repeat replays the full Insert session, including cursor-moves. Undo and quote. still restart at each cursor-move. Use i_CTRL-O to get the old behavior:

```
inoremap <Left>  <C-o><Left>
inoremap <Right> <C-o><Right>
```

- visual-repeat is "semantic" instead of "fixed-size". See visual-fixed-size if you prefer Vim's "fixed-size" Visual-repeat.
- :diffget, :diffput (do, dp) can be . repeated.
- Q adds a multicursor instead of replaying the last recorded register. To get the old behavior:

```
vim.keymap.set('n', 'Q', function()
  local reg = vim.fn.reg_recorded()
  return reg == '' and '' or ('@' .. reg)
end, { expr = true })
```

EVENTS

- vim.on_key() callbacks are not invoked for keys consumed by getchar().
- BufModifiedSet has been removed. Use the OptionSet event with pattern "modified" instead.
- ui-messages `msg_show.bufwrite` and `msg_show.completion` messages are now `msg_show.progress` events.

LSP

- `vim.lsp.ClientConfig.cmd` given as a string array now uses `vim.lsp.ClientConfig.root_dir` as its default process working directory.
- `client.attached_buffers[buf]` now stores `languageId` string (was boolean).

LUA

- "standalone" Lua interpreter mode `nvim -ll` was removed. Use -l script mode instead.
- vim.opt no longer supports chaining multiple infix operators (e.g. `vim.opt.wildignore + '*.o' + '*.obj'`). Instead, use tables: `vim.opt.wildignore + {'*.o', '*.obj'}`

OPTIONS

- The 'statusline' %= item is no longer ignored in item groups. It applies to the containing item group and its `minwid`. An %< item inside an item group now only applies to the containing group and its `maxwid`. Therefore, multiple %< can now be effective. The default truncation of item groups is now at the first contained item. To restore the previous truncation from the left, add a %< to the start. The same applies to 'rulerformat', 'statuscolumn', 'tabline', 'winbar', 'titlestring', and 'iconstring'.
- 'rulerformat' can no longer be set empty, the new default expression will be restored instead.

PLUGINS

- todo

TREESITTER

- todo

UI

- The bottom right cell of the screen is no longer left empty when the ruler is shown. This was originally intended to prevent unwanted scrolling on quirky terminals.

VIMSCRIPT

- Removed: ctxget(), ctxpop(), ctxpush(), ctxset(), ctxsize(). Use nvim_get_context() and nvim_load_context() instead.

NEW FEATURES

news-features

The following new features were added.

API

- nvim_win_resize() resizes a window with a specified anchor edge. nvim_win_set_height() and nvim_win_set_width() are now deprecated.
- api-contract treats `opts` and any parameters thereafter, as optional (equivalent to passing an empty dict). Clients may omit such optional parameters.
  - The Lua vim.api bridge is also a "client" and thus may omit `opts` and any following parameters:

    ```
    -- Before:
    vim.print(vim.api.nvim_get_mark('A', {}))
    -- After:
    vim.print(vim.api.nvim_get_mark('A'))
    ```

- nvim_buf_call() and nvim_win_call() now preserve multiple return values.
- nvim_del_keymap(), nvim_buf_del_keymap() and vim.keymap.del() can match only `{lhs}`, not `{rhs}`, with `opts.lhs=true`.
- nvim_set_hl() supports "font" key.
- nvim_open_win() `zindex` controls whether the UI will use a dimmed cursor shape when an unfocused float is on top of the cursor.
- nvim_echo() distinguishes zero percent from omitted percent for Progress events.
- nvim_create_user_command() accepts `desc` for Vimscript commands.
- Support for tabpage-local options ('cmdheight'):
  - gettabvar()
  - gettabwinvar()
  - nvim_get_option_info2()
  - nvim_get_option_value()
  - nvim_set_option_value()
- nvim_buf_set_extmark() `virt_lines_overflow` accepts "wrap" to enable wrapping onto extra rows and "auto" which enables horizontal scrolling when 'nowrap' is set and wrapping when 'wrap' is set.
- nvim_buf_set_extmark() `virt_lines` can extend the last `hl` in each `virt_line` till the end of the line.
- nvim_set_option_value() accepts a new `operation` field to modify the existing option value.
- nvim_set_option_value() returns the new option value.
- nvim_create_autocmd() is now api-fast, so it can be called from a fast event context (e.g. vim.uv callbacks).
- nvim_create_autocmd() callback event-args provide `win`.
- nvim_chan_set() sets channel properties. With `detach=false`, any RPC channel can enable Nvim "self-exit on channel-close" behavior.

BUILD

- Building using "zig build" requires zig 0.16.x.
- zig build: "-Dcross" option was removed. Often cross-compilation is now detected, so e.g. from a linux host:

  ```
  zig build -Dtarget=aarch64-macos
  ```

  will automatically compile a host Lua for use during build.
- zig build: The new "-Dhost={target_string}" option can be used to override the used host. Use "-Dhost=native" to force cross-compiling or "-Dhost=" (empty string) to assume that target binaries can run on the host during the build process (e.g. if target is x86 on a x86_64 system, or if emulation set up with binfmt or similar).
- zig build: Nvim can now be built to produce a WebAssembly binary via "-Dtarget=wasm32-emscripten", which can run inside a web browser. A basic browser-based demo for running it is also included. Still very experimental.

DEFAULTS

- 'rulerformat' default is exposed as a statusline expression (previously it was implemented as an internal C routine).

DIAGNOSTICS

- vim.diagnostic.status() uses the severity names defined by the `signs` field of vim.diagnostic.config(), if any.
- `virtual_lines.overflow` in vim.diagnostic.config() controls how virtual lines wider than the window are displayed.

EDITOR

- Multiple cursors (multicursor): edit in many places at once.
  - Toggle cursors with Q, <C-LeftMouse>, or via API (the "nvim.multicursor" extmarks namespace).
  - Toggle "follow mode" with g=.
  - Clear cursors with CTRL-L, restore them with gQ.
  - zq places a cursor at all targets of a motion. E.g. "zqgn" places at every match (and "gv" selects the matches), "Vzqw" places at every Visual-selected word.
- :bcd sets a buffer-local directory. getcwd() and haslocaldir() take `bufnr` as a third parameter.
- :lcd! :tcd! :bcd! unsets the directory of that scope.
- :command-completion-customlist can return a list of dictionaries with kind/menu/info/abbr for the popup menu.
- cmdwin (q:, q/, q?, c_CTRL-F) is implemented as a "normal" buffer+window instead of a nested-state modal loop:
  - You can create/navigate windows/tabpages while cmdwin is active. Chaos!
  - 'inccommand' works in cmdwin!
  - API calls (e.g. nvim_buf_delete()) that previously failed with "E11: Invalid in command-line window" while cmdwin was open, now work normally. Async plugins no longer need special CmdwinLeave workarounds.
  - The cmdwin-char is shown via 'statuscolumn'.
- gf and <cfile> support `file://…` URIs.
- :help files render tables (example: api-types).
  - Spec: https://github.com/neovim/tree-sitter-vimdoc#tables
- :help documentation is available as PDF via: https://neovim.io/doc/user/nvim-help.pdf
- :log opens log files.
- :marks can list extmarks for a given namespace.

- :marks gained tab-completion.
- :restart saves/restores the current session (window layout, buffers, …).
  - :restart! (with a bang "!") does not save/restore the session.
- ZR restarts Nvim (:restart).
- :detach with range "%" detaches all UIs except the current one.
- :detach! (with bang "!") marks the current UI as detachable: Nvim will continue running if the UI disconnects unexpectedly (e.g. if you accidentally close your terminal, ssh connection lost, etc.). Use :connect to reattach.
- :uptime displays uptime.
- :packupdate and :packdel for managing vim.pack.
- 'scrollback' is now also valid in prompt-buffer buffers to limit the number of history lines kept above the prompt.
- v_al and v_il text objects select the whole buffer and the current line without leading or trailing white space.
- 'autoread' uses file system watchers to detect external changes in real-time, instead of only on FocusGained or :checktime.
- During complete()-triggered completion, CTRL-N and CTRL-P are now subject to insert-mode mappings.
- undo restores the cursor to its original position ("diw", "atest<Esc>", "d^", …), instead of jumping to the start of the changed text.
- undo no longer restores the old (wrong) position of a mark that you moved (m, :mark) later. The mark shifts with the text it was moved to, the same as a mark the change never touched.
- Multibyte characters, translated by 'langmap', now invoke correct mappings. Example:

```
set langmap=ő]
```

  Typing őő in Normal-mode now invokes a mapping bound to]].
- New complete-items field "commit_chars": characters that accept the selected completion item when typed.

EVENTS

- :delmarks now triggers the MarkSet autocommand with line==col==0, same as nvim_buf_del_mark()
- ChanClose is triggered after a channel is closed, before it is removed.
- CmdAtom fires after each atomic user action (motion, mapping, operator, insert session, visual, Ex command, …).
- SessionWritePre event emits just before :mksession.
- TextPutPre and TextPutPost are triggered before/after putting text.
- TabMoved is triggered when tabs are reordered.
- DirReadPost is triggered after each dir listing is rendered, so it can be sorted or filtered. See dir-render.

HIGHLIGHTS

- Dimmed for text that should be de-emphasized.
- vim.hl.hl_op() highlights text regions for TextYankPost and TextPutPost events. It replaces the now deprecated `vim.hl.on_yank()`.

LSP

- LSP capabilities:
  - Completion supports `CompletionItem.preselect` if 'completeopt' has "preselect". https://microsoft.github.io/language-server-protocol/specification/#completionClientCapabilities
  - `textDocument/foldingRange` vim.lsp.foldtext() highlights collapsed text. https://microsoft.github.io/language-server-protocol/specification/#textDocument_foldingRange
  - Completion supports `CompletionList.applyKind`: a server can ask for `commitCharacters` and `data` from `itemDefaults` to be merged with the item's own values instead of replacing them. https://microsoft.github.io/language-server-protocol/specification/#completionItemApplyKinds
  - Completion supports a `completionProvider` registered with `client/registerCapability` instead of declared at `initialize`. Nvim does not advertise `completion.dynamicRegistration` by default; set it in vim.lsp.Config `capabilities` to opt in. https://microsoft.github.io/language-server-protocol/specification/#client_registerCapability
- `:checkhealth vim.lsp` highlights the "current buffer".
- vim.lsp.buf.declaration(), vim.lsp.buf.definition(), vim.lsp.buf.type_definition(), and vim.lsp.buf.implementation() now follows 'switchbuf'.
- vim.lsp.buf.declaration(), vim.lsp.buf.definition(), vim.lsp.buf.type_definition(), and vim.lsp.buf.implementation() now supports passing arbitrary positions.
- Support for nested snippets.
- lsp-completion supports `commitCharacters`: typing a commit character while a completion item is selected accepts the item as with complete_CTRL-Y, then inserts the character. Previously `commitCharacters` had no effect.
- The `reuse_client` predicate now passes the target buffer. See vim.lsp.Config and vim.lsp.start()
- vim.lsp.formatexpr() now falls back to `textDocument/formatting` if the whole buffer is being formatted and the language server doesn't support `textDocument/rangeFormatting`.
- vim.lsp.buf.format() respects an explicit `timeout_ms` with `async=true`. The timeout applies per client. Async requests have no timeout when the option is omitted.
- Support for `workspace/foldingRange/refresh`: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.18/specification/#workspace_foldingRange_refresh
- vim.lsp.completion.get() accepts a `client_id` to request from only the given clients.

LUA

- lua-async: `vim.async` provides structured concurrency primitives for Lua code, including task handles, await helpers, cooperative cancellation, completion-order iteration, and semaphores.
- vim.ui.img can display images. Use `:checkhealth vim.health` to confirm your terminal supports it.
- vim.net.request() can specify custom headers by passing `opts.headers`.
- vim.net.request() can now accept `method` param overload for multiple HTTP methods.
- writefile() treats Lua and RPC strings as Blob, so it can be used to write binary data.
- vim.fs.abspath() gained `cwd` and `plain` parameters.
- vim.fs.dir() with `opts.err=true`, reports errors. An inaccessible root dir yields a single (name, nil, err) item.
- vim.fs.dir() gained a `plain` parameter.

- vim.fs.find() returns a list of errors as its second return value.
- vim.fs.mkdir() creates directories, including parent directories with `opts.parents=true`.
- vim.fs.slug() generates a filesystem-safe file name from an arbitrary identity string, with an optional length bound.
- vim.filetype.inspect() returns a copy of the internal tables used for filetype detection.
- Added `__eq` metamethod to vim.VersionRange. 2 distinct but representing the same range instances now compare equal.
- Documentation for vim.fn now points to preferred Lua alternatives when available.
- vim.list.unique() and vim.list.bisect() now support passing a string as a shorthand of a key
- vim.isnil() tests if a value is `nil` or vim.NIL.
- vim.nonnil() returns the first argument which is not nil.
- vim.npcall() calls the function `fn` in protected-mode like pcall(), but returns `nil` on error.
- vim.pos can now convert between positions and buffer offsets.
- vim.pos and vim.range can now convert between mark positions.
- vim.pos.cursor() without arguments returns the cursor position of the current window.
- vim.ui.input() now allows setting input scope.
- vim.log provides a logging interface.
- vim.pack.get() output includes revision of a pending update.
- vim.pack.get() can fetch new updates before computing the output.
- vim.pack supports vim.pack-manifest of plugins.
- vim.o now accepts table style values for assignment.
- vim.keycode() returns structured info as return value 2.
- Iter:count() counts items in the iterator.

OPTIONS

- Lua functions/closures can be assigned to "func" and "expr" options: option-value-function (e.g. `vim.o.operatorfunc`), expr-option-function (e.g. `vim.wo.foldexpr`, `vim.bo.indentexpr`). Reading such an option:
  - ...from Lua, gets the function itself.
  - ...from Vimscript/RPC, gets a human-readable "<Lua …>" hint string.
- 'ttyfast' can be disabled during startup by setting $NVIM_NOTTYFAST.
- 'scrolloffpad' allows vertically centering cursor at the end of file.
- 'shortmess' flag shm-u silences undo/redo messages.
- 'statusline' supports stl-%0{ to insert the expression result verbatim. The %= item is supported within item groups to separate alignment sections within the containing group until its `minwid` is reached. The truncation of an item group to its `maxwid` can be controlled with the %< item. The new %#(...%#) item denotes a highlight scope. The same applies to 'rulerformat', 'statuscolumn', 'tabline', 'winbar', 'titlestring', and 'iconstring'.
- 'winpinned' prevents window from closing unless specifically targeted.
- 'packlockfile' sets the path used for vim.pack-lockfile.
- 'previewpopup' decides how the preview window is displayed.
- 'messagesopt' items "pager" and "timeout" configure the ui2 message pager key and how long a message stays in the message window.

PERFORMANCE

- LSP shares identical file watcher registrations, reducing startup work in large workspaces.
- treesitter-highlight performance on large injection-heavy files improves by 50% to 100% by reusing edited child-tree ranges.
- Nvim architecture allows pure-Lua implementations of some `vim.fn` functions, which skips the Vimscript <=> Lua "bridge" (no data conversion/marshalling) entirely, if the `vim.fn` function is called from Lua.
- Internal LSP data is cleared using `table.clear`, which reduces GC and memory reallocation during each data reset.
- RPC client avoids string allocations when parsing Content-Length messages.
- LSP: "overscan" semantic_token range requests to avoid flicker.
- vim.hl.range() and vim.hl.hl_op() are faster for large characterwise and linewise regions.
- The display of multibyte text with grapheme clusters has been optimized, by calculating byte length and cell width of clusters in a single loop.

PLUGINS

- The zip plugin is now zip, a read-only archive browser built on dir. To use the legacy plugin instead:

```
:packadd old-zip
```

- provider: add bun support for Node.js plugins

STARTUP

- --embed with --listen waits for a UI on the listen address if stdin is not a pipe (e.g. tty or /dev/null), instead of exiting.

TERMINAL

- todo

TREESITTER

- vim.treesitter.get_captures() returns highlight captures at a given position, on a whole line, or overlapping a range. It replaces vim.treesitter.get_captures_at_pos().
- v_]N v_[N expand selection to sibling treesitter node.
- treesitter-highlight-conceal can be removed by adding a `@noconceal` capture.
- vim.treesitter.select() starts or adjusts a visual selection at cursor, based on tree nodes.
- The `diff` treesitter parser is bundled.

TUI

- The TUI will re-query the terminal's background color when resuming from a suspended state, and Nvim will update 'background' accordingly.
- User can override the builtin "terminfo" via $NVIM_TERMDEFS. This is similar to Vim's t_xx options.

UI

- :checkhealth shows filewatcher info in the Performance section.
- Floating window border highlights (hl-FloatBorder, hl-FloatTitle, hl-FloatFooter) combine with the window-local hl-Normal background instead of falling back to the global background.
- These builtin "picker" menus delegate to vim.ui.select():
  - :browse oldfiles
  - :recover
  - :tselect
  - z= (spell suggest)
- The ui2 dialog window scrolls with the mouse wheel, by 'mousescroll' lines.
- The ui2 `[+x]` "spill" indicator is highlighted with hl-MoreMsg.

VIMSCRIPT

- `fnamemodify(':h')` preserves logical roots for more path formats.
- v:exitreason is set before QuitPre.
- v:startreason indicates whether Nvim started normally or by :restart.
- v:starttime is the process start time (nanoseconds since UNIX epoch).
- v:useractive indicates user activity.
- serverlist() with `info=true` returns details for each server (own + peers).
- These Vimscript functions may be called from api-fast context (e.g. vim.uv timer/handle callbacks): keytrans(), char2nr(), nr2char(), str2list(), strpart(), strcharpart(), strgetchar(), strchars(), strcharlen(), strlen(), strdisplaywidth(), byteidx(), charidx(), utf16idx(), tr(), trim().

CHANGED FEATURES news-changed

These existing features changed their behavior.

- Environment variable name (`$NAME`) validity is no longer controlled by 'isident'.
- `nvim_exec_autocmds({buf=…})` runs in the context of the target buffer.
- SessionLoadPost is no longer included in views saved by :mkview.
- OptionSet is no longer triggered during startup by automatic 'background' detection.
- SessionWritePre and SessionWritePost are no longer triggered by :mkview, :mkvimrc, or :mkexrc.
- Editing a local directory now shows its contents in a `filetype=directory` buffer. See dir.
- :Open with no arguments uses the current file.
- The "buffer" key was renamed to "buf" in these functions (but the old name "buffer" is still accepted, for backwards compatibility):
  - nvim_clear_autocmds()
  - nvim_create_autocmd()
  - nvim_exec_autocmds()
  - nvim_get_autocmds()
- vim.keymap.set() and vim.keymap.del() accept a list of strings for `lhs`.
- 'statuscolumn' click function labels can be different across rows. v:virtnum and v:lnum are available during the click function callback.
- Markdown inline highlighting now conceals the backslash in backslash escapes.
- Markdown inline backslash escapes and hard line breaks no longer use the `@string.escape` capture.
- vim.lsp.util.open_floating_preview() windows (e.g. vim.lsp.buf.hover()) converted to normal windows (e.g. with CTRL-W_H) are no longer closed automatically.
- vim.fs.normalize() `opts.expand_env=false` key was renamed to `opts.plain=true` and now does not expand leading tildes ("~") in addition to environment variables ("expand_env" is still accepted, for backwards compatibility).
- :helptags finds help tags with the "vimdoc" treesitter parser (and thus requires it to be installed).
- `:helptags ALL` reports E152 for "doc" directories it cannot write, instead of silently skipping them.
- The highlighting groups hl-CursorLineFold and hl-CursorLineSign are always used when 'cursorline' is set.
- vim.lsp.util.apply_text_edits() accepts `opts.keep_cursor`, to hold the cursor on the text it is on while edits move around it.
- 'tagfunc' is now also used for :tag completion.
- Some character clusters are restored to occupy two cells like they did prior to Nvim 0.11. These include Thai syllables like □□□ and half-width Katakana with voicing marks □□. The TUI renders these in a safe way which also supports terminals which consider these to be single-width.
- in script mode -l, print() now behaves more similar to standalone lua. Printing is done to stdout, and empty lines are not stripped. No translation of special characters is done, so escape sequences can be used when printing to a terminal.
- It is no longer allowed to change the window layout during a DiffUpdated autocommand.

REMOVED FEATURES news-removed

These deprecated features were removed.

- Vimscript functions ctxget(), ctxpop(), ctxpush(), ctxset(), ctxsize() and the "context stack". Use nvim_get_context() and nvim_load_context() with a script-owned list instead.
- vim.lsp.buf.declaration(), vim.lsp.buf.definition(), vim.lsp.buf.type_definition(), and vim.lsp.buf.implementation() no longer accept the `reuse_win` option.

DEPRECATIONS

news-deprecations

See deprecated-0.13.

Nvim neovim nvim-intro

Nvim is based on Vim by Bram Moolenaar. Nvim is emphatically a fork of Vim, not a clone: compatibility with Vim (especially editor and Vimscript features, except Vim9script) is maintained where possible. See vim-differences for the complete reference.

- If you already use Vim, see nvim-from-vim for a quickstart.
- If you have never used Vim or Nvim before, see below.

What now? nvim-quickstart

To learn how to use Vim in 30 minutes, try the tutorial:

```
:Tutor<Enter>
```

Or watch this 10-minute video: https://youtu.be/TQn2hJeHQbM .

config-example example-init To customize Nvim, you will need a config file. Create your init.lua by copying the "example_init.lua" file:

```
:exe 'edit' stdpath('config') .. '/init.lua'
:read $VIMRUNTIME/example_init.lua
:write ++p
:restart
```

See lua-guide for practical notes on using Lua to configure Nvim.

"IDE" features in Nvim are provided by LSP.

If you are just trying out Nvim for a few minutes, and want to see the extremes of what it can do, try one of these popular "extension packs" or "distributions" (**Note:** Nvim is not affiliated with these projects, and does not support them):

- lazyvim https://www.lazyvim.org/
- nvchad https://nvchad.com/
- kickstart https://github.com/nvim-lua/kickstart.nvim
  - Not recommended; use example-init instead.

However, we recommend (eventually) taking time to learn Nvim from its stock configuration, and incrementally setting options and adding plugins to your config as you discover a need.

Transitioning from Vim nvim-from-vim

1. To start the transition, create your init.vim (user config) file:

   ```
   :exe 'edit' stdpath('config') .. '/init.vim'
   :write ++p
   ```

2. Add these contents to the file:

   ```
   set runtimepath^=~/.vim runtimepath+=~/.vim/after
   let &packpath = &runtimepath
   source ~/.vimrc
   ```

3. Restart Nvim, your existing Vim config will be loaded.

   ```
   :restart
   ```

   See provider-python and provider-clipboard for additional software you might need to use some features.

Your Vim configuration might not be entirely Nvim-compatible (see vim-differences). For example the 'ttymouse' option was removed from Nvim, because mouse support is always enabled if possible. If you use the same vimrc for Vim and Nvim you could guard 'ttymouse' in your configuration like so:

```
if !has('nvim')
  set ttymouse=xterm2
endif
```

And for Nvim-specific configuration, you can do this:

```
if has('nvim')
  tnoremap <Esc> <C-\><C-n>
endif
```

For a more granular approach use exists():

```
if exists(':tnoremap')
  tnoremap <Esc> <C-\><C-n>
endif
```

Now you should be able to explore Nvim more comfortably. Check nvim-features for more information.

portable-config

Because Nvim follows the XDG base-directories standard, configuration on Windows is stored in ~/AppData instead of ~/.config. But you can still share the same Nvim configuration on all of your machines, by creating ~/AppData/Local/nvim/init.vim containing just this line:

```
source ~/.config/nvim/init.vim
```

Options

For an overview of options see quickref.txt option-list.

Vim has a number of internal variables and switches which can be set to
achieve special effects. These options come in three forms:

boolean	can only be on or off	boolean toggle
number	has a numeric value	
string	has a string value	

1. Setting options set-option E764

:se :set

:se[t][!] Show all options that differ from their default value.
When [!] is present every option is on a separate
line.

:se[t][!] all Show all options.
When [!] is present every option is on a separate
line.

E518 E519

:se[t] {option}? Show value of {option}.
NOTE: some legacy options were removed. nvim-removed

:se[t] {option} Toggle option: set, switch it on.
Number option: show value.
String option: show value.

:se[t] no{option} Toggle option: Reset, switch it off.

:set-! :set-inv

:se[t] {option}! or
:se[t] inv{option} Toggle option: Invert value.

:set-default :set-& :set-&vi :set-&vim

:se[t] {option}& Reset option to its default value.
:se[t] {option}&vi Reset option to its Vi default value.
:se[t] {option}&vim Reset option to its Vim default value.

:se[t] all& Set all options to their default value. The values of
these options are not changed:
'columns'
'lines'
Warning: This may have a lot of side effects.

:set-args :set= E487 E521

:se[t] {option}={value} or
:se[t] {option}:{value}
Set string or number option to {value}.
For numeric options the value can be given in decimal,
hex (preceded with 0x) or octal (preceded with '0' or
'0o').
The old value can be inserted by typing 'wildchar' (by
default this is a <Tab>). Many string options with
fixed syntax also support completing known values.
See cmdline-completion and complete-set-option.
White space between {option} and '=' is allowed and
will be ignored. White space between '=' and {value}
is not allowed.
See option-backslash for using white space and
backslashes in {value}.

:se[t] {option}+={value} :set+=

Add the {value} to a number option, or append the
{value} to a string option. When the option is a
comma-separated list, a comma is added, unless the
value was empty.
If the option is a list of flags, superfluous flags
are removed. When adding a flag that was already
present the option value doesn't change.
When the option supports "key:value" items and {value}
contains a "key:value" item or multiple
comma-separated items, each item is processed
individually:

- A "key:value" item where the key already exists with

a different value: the old item is removed and the
new item is appended to the end.

- A "key:value" item that is an exact duplicate is

left unchanged.

- Other items that already exist are left unchanged.

- New items are appended to the end.

Also see :set-args above.

:se[t] {option}^={value} :set^=

Multiply the {value} to a number option, or prepend

the {value} to a string option. When the option is a
comma-separated list, a comma is added, unless the
value was empty.
When the option supports "key:value" items and {value}
contains a "key:value" item or multiple
comma-separated items, each item is processed
individually. Works like :set+= but new items are
prepended to the beginning instead of appended.
Also see :set-args above.

:se[t] {option}-={value} :set-=

Subtract the {value} from a number option, or remove
the {value} from a string option, if it is there.
If the {value} is not found in a string option, there
is no error or warning. When the option is a comma-
separated list, a comma is deleted, unless the option
becomes empty.
When the option is a list of flags, {value} must be
exactly as they appear in the option. Remove flags
one by one to avoid problems.
When the option supports "key:value" items and {value}
contains a "key:value" item or multiple
comma-separated items, each item is processed
individually. A "key:value" item removes the existing
item with that key regardless of its value. A "key:"
item also removes by key match.
The individual values from a comma separated list or
list of flags can be inserted by typing 'wildchar'.
See complete-set-option.
Also see :set-args above.

The {option} arguments to ":set" may be repeated. For example:

```
:set ai nosi sw=3 ts=3
```

If you make an error in one of the arguments, an error message will be given
and the following arguments will be ignored.

:set-verbose

When 'verbose' is non-zero, displaying an option value will also tell where it
was last set. Example:

```
:verbose set shiftwidth cindent?
```

shiftwidth=4
Last set from modeline line 1
cindent
Last set from /usr/local/share/vim/vim60/ftplugin/c.vim line 30
This is only done when specific option values are requested, not for ":verbose
set all" or ":verbose set" without an argument.
When the option was set by hand there is no "Last set" message.
When the option was set while executing a function, user command or
autocommand, the script in which it was defined is reported.
A few special texts:
Last set from modeline line 1
		Option was set in a modeline.
Last set from --cmd argument
		Option was set with command line argument --cmd or +.
Last set from -c argument
		Option was set with command line argument -c, +, -S or
		-q.
Last set from environment variable
		Option was set from $VIMINIT.
Last set from error handler
		Option was cleared when evaluating it resulted in an error.

option-backslash

To include white space in a string option value it has to be preceded with a
backslash. To include a backslash you have to use two. Effectively this
means that the number of backslashes in an option value is halved (rounded
down).
In options 'path', 'cdpath', and 'tags', spaces have to be preceded with three
backslashes instead because they can be separated by either commas or spaces.
Comma-separated options like 'backupdir' and 'tags' will also require commas
to be escaped with two backslashes, whereas this is not needed for
non-comma-separated ones like 'makeprg'.
When setting options using :let and literal-string, you need to use one
fewer layer of backslash.
A few examples:

```
:set makeprg=make\ file     results in "make file"
:let &makeprg='make file'           (same as above)
:set makeprg=make\\\ file           results in "make\ file"
:set tags=tags\ /usr/tags           results in "tags" and "/usr/tags"
:set tags=tags\\\ file      results in "tags file"
:let &tags='tags\ file'     (same as above)

:set makeprg=make,file      results in "make,file"
:set makeprg=make\\,file            results in "make\,file"
:set tags=tags,file                 results in "tags" and "file"
:set tags=tags\\,file       results in "tags\,file"
:let &tags='tags\,file'     (same as above)
```

The "|" character separates a ":set" command from a following command. To include the "|" in the option value, use "\|" instead. This example sets the 'titlestring' option to "hi|there":

```
:set titlestring=hi\|there
```

This sets the 'titlestring' option to "hi" and 'iconstring' to "there":

```
:set titlestring=hi|set iconstring=there
```

Similarly, the double quote character starts a comment. To include the '"' in the option value, use '\"' instead. This example sets the 'titlestring' option to "hi "there"":

```
:set titlestring=hi\ \"there\"
```

For Win32 backslashes in file names are mostly not removed. More precise: For options that expect a file name (those where environment variables are expanded) a backslash before a normal file name character is not removed. But a backslash before a special character (space, backslash, comma, etc.) is used like explained above.
There is one special situation, when the value starts with "\\":

```
:set dir=\\machine\path     results in "\\machine\path"
:set dir=\\\\machine\\path          results in "\\machine\path"
:set dir=\\path\\file       results in "\\path\file" (wrong!)
```

For the first one the start is kept, but for the second one the backslashes are halved. This makes sure it works both when you expect backslashes to be halved and when you expect the backslashes to be kept. The third gives a result which is probably not what you want. Avoid it.

add-option-flags remove-option-flags

E539

Some options are a list of flags. When you want to add a flag to such an option, without changing the existing ones, you can do it like this:

```
:set guioptions+=a
```

Remove a flag from an option like this:

```
:set guioptions-=a
```

This removes the 'a' flag from 'guioptions'.
Note that you should add or remove one flag at a time. If 'guioptions' has the value "ab", using "set guioptions-=ba" won't work, because the string "ba" doesn't appear.

:set_env expand-env expand-environment-var

Environment variables in specific string options will be expanded. If the environment variable exists the '$' and the following environment variable name is replaced with its value. If it does not exist the '$' and the name are not modified. Any non-id character (not a letter, digit or '_') may follow the environment variable name. That character and what follows is appended to the value of the environment variable. Examples:

```
:set term=$TERM.new
:set path=/usr/$INCLUDE,$HOME/include,.
```

When adding or removing a string from an option with ":set opt-=val" or ":set opt+=val" the expansion is done before the adding or removing.

Handling of local options local-options

Note: The following also applies to global-local options.

Some of the options only apply to a window or buffer. Each window or buffer has its own copy of this option, thus each can have its own value. This allows you to set 'list' in one window but not in another. And set 'shiftwidth' to 3 in one buffer and 4 in another.

The following explains what happens to these local options in specific situations. You don't really need to know all of this, since Vim mostly uses the option values you would expect. Unfortunately, doing what the user expects is a bit complicated...

When splitting a window, the local options are copied to the new window. Thus right after the split the contents of the two windows look the same.

When editing a new buffer, its local option values must be initialized. Since the local options of the current buffer might be specifically for that buffer, these are not used. Instead, for each buffer-local option there also is a global value, which is used for new buffers. With ":set" both the local and global value is changed. With "setlocal" only the local value is changed, thus this value is not used when editing a new buffer.

When editing a buffer that has been edited before, the options from the window that was last closed are used again. If this buffer has been edited in this window, the values from back then are used. Otherwise the values from the last closed window where the buffer was edited last are used.

":setlocal" can be used to set a local window option specifically for a type of buffer. When you edit another buffer in the same window, you don't want to keep using these local window options. Meanwhile ":set" also sets a global value of a local window option, which is used when editing another buffer. Each window has its own copy of these global values, making them local to the window, but global to all buffers in the window. With this you can do:

```
:e one
:set list
:e two
```

Now the 'list' option will also be set in "two", since with the ":set list" command you have also set the global value.

```
:set nolist
:e one
:setlocal list
:e two
```

Now the 'list' option is not set, because ":set nolist" resets the global value, ":setlocal list" only changes the local value and ":e two" gets the global value. Note that if you do this next:

```
:e one
```

You will get back the 'list' value as it was the last time you edited "one". The options local to a window are remembered for each buffer. This also happens when the buffer is not loaded, but they are lost when the buffer is wiped out :bwipe.

Special local window options local-noglobal

The following local window options won't be copied over when new windows are created, thus they behave slightly differently:

Option	Reason
'previewwindow'	there can only be a single one
'scroll'	specific to existing window
'winfixbuf'	specific to existing window
'winfixheight'	specific to existing window
'winfixwidth'	specific to existing window
'winpinned'	specific to existing window

Special local buffer options

The following local buffer options won't be copied over when new buffers are created, thus they behave slightly differently:

Option	Reason
'filetype'	explicitly set by autocommands
'syntax'	explicitly set by autocommands
'bufhidden'	denote special-buffers
'buftype'	denote special-buffers
'readonly'	will be detected automatically
'modified'	will be detected automatically

:setl :setlocal

```
:setl[ocal][!] ...      Like ":set" but set only the value local to the
                        current buffer or window.  Not all options have a
                        local value.  If the option does not have a local
                        value the global value is set.
                        With the "all" argument: display local values for all
                        local options.
                        Without argument: Display local values for all local
                        options which are different from the default.
                        When displaying a specific local option, show the
```

local value. For a global/local boolean option, when
the global value is being used, "--" is displayed
before the option name.
For a global option the global value is
shown (but that might change in the future).

:se[t] {option}< Set the effective value of {option} to its global
value.
For global-local options, the local value is removed,
so that the global value will be used.
For all other options, the global value is copied to
the local value.

:setl[ocal] {option}< Set the effective value of {option} to its global
value by copying the global value to the local value.

Note that the behaviour for global-local options is slightly different
between string and number-based options.

:setg :setglobal

:setg[lobal][!] ... Like ":set" but set only the global value for a local
option without changing the local value.
When displaying an option, the global value is shown.
With the "all" argument: display global values for all
local options.
Without argument: display global values for all local
options which are different from the default.

For buffer-local and window-local options:

Command	global value	local value	condition
:set option=value	set	set	
:setlocal option=value	-	set	
:setglobal option=value	set	-	
:set option?	-	display	local value is set
:set option?	display	-	local value is not set
:setlocal option?	-	display	
:setglobal option?	display	-	

Global options with a local value global-local

Options are global when you mostly use one value for all buffers and windows.
For some global options it's useful to sometimes have a different local value.
You can set the local value with ":setlocal". That buffer or window will then
use the local value, while other buffers and windows continue using the global
value.

For example, you have two windows, both on C source code. They use the global
'makeprg' option. If you do this in one of the two windows:

```
:set makeprg=gmake
```

then the other window will switch to the same value. There is no need to set
the 'makeprg' option in the other C source window too.
However, if you start editing a Perl file in a new window, you want to use
another 'makeprg' for it, without changing the value used for the C source
files. You use this command:

```
:setlocal makeprg=perlmake
```

You can switch back to using the global value by making the local value empty:

```
:setlocal makeprg=
```

This only works for a string option. For a number or boolean option you need
to use the "<" flag, like this:

```
:setlocal autoread<
```

Note that for non-boolean and non-number options using "<" copies the global
value to the local value, it doesn't switch back to using the global value
(that matters when the global value changes later). You can also use:

```
:set path<
```

This will make the local value of 'path' empty, so that the global value is
used. Thus it does the same as:

```
:setlocal path=
```

Note: In the future more global options can be made global-local. Using
":setlocal" on a global option might work differently then.

option-value-function

Some options ('completefunc', 'findfunc', 'omnifunc', 'operatorfunc',
'quickfixtextfunc', 'tagfunc' and 'thesaurusfunc') can be set to a function
name, a Funcref, a lambda, or a Lua function.
Examples:

```
set opfunc=MyOpFunc
set opfunc=function('MyOpFunc')
set opfunc=funcref('MyOpFunc')
set opfunc={a\ ->\ MyOpFunc(a)}
```

Set to a script-local function:

```
set opfunc=s:MyLocalFunc
set opfunc=<SID>MyLocalFunc
```

Set using a funcref variable:

```
let Fn = function('MyTagFunc')
let &tagfunc = Fn
```

Set using a lambda expression:

```
let &tagfunc = {t -> MyTagFunc(t)}
```

Set using a variable with lambda expression:

```
let L = {a, b, c -> MyTagFunc(a, b , c)}
let &tagfunc = L
```

Set to a Lua function (from Lua):

```
vim.o.operatorfunc = function(motion) … end
```

The value read back from the option reflects how it was set:

- A Function name or Funcref reads back as the function name.
- A Vimscript lambda or partial reads back as an evaluatable string, e.g. function('<lambda>2') or function('MyFunc', [10]).
- A script-local ("s:") name is expanded to <SNR> form.
- A Lua function is returned as itself from Lua, or as a "<Lua …>" hint from Vimscript/RPC.
  - **Note:** Lua functions cannot be serialized, so it cannot be saved/restored by :mksession :mkview :mkvimrc.

Calling a function in an expr option expr-option-function

The value of an "expr" option ('charconvert', 'diffexpr', 'foldexpr', 'foldtext', 'formatexpr', 'includeexpr', 'indentexpr', 'patchexpr') is a Vimscript expression that is evaluated to get a value. Context is passed in v: variables, e.g. 'foldexpr' is evaluated for each line with v:lnum set.

The evaluation can have quite a bit of overhead. One way to minimize the overhead, and also to keep the option value very simple, is to define a function and set the option to call it without arguments. A v:lua-call can also be used. Example:

```
lua << EOF
  function _G.MyFoldFunc()
    -- ... compute fold level for line v:lnum
    return level
  end
EOF
set foldexpr=v:lua.MyFoldFunc()
```

From Lua the option can also be set directly to a function, which is called with no arguments (read the context from v: variables):

```
vim.wo.foldexpr = function() return vim.v.lnum == 1 and '>1' or '1' end
```

The value read back from the option follows the same rules as option-value-function: a string reads back as itself, and a Lua function reads back as the function (from Lua) or a "<Lua …>" hint (Vimscript/RPC).

Setting the filetype

:setf[iletype] [FALLBACK] {filetype} :setf :setfiletype

Set the 'filetype' option to {filetype}, unless it was already set (non-empty) in a sequence of (nested) autocommands.
This is short for:

```
:if !did_filetype()
:  setlocal filetype={filetype}
:endif
```

Used in a filetype.vim file to avoid setting the 'filetype' option twice, causing different settings and syntax files to be loaded.

When the optional FALLBACK argument is present, a later :setfiletype command will override the 'filetype'. This is to be used for filetype detections that are just a guess. did_filetype() will return false after this command.

option-window optwin
:bro[wse] se[t] :set-browse :browse-set :opt :options
:opt[ions] Open a window for viewing and setting all options.
Options are grouped by function.
Offers short help for each option. Hit <CR> on the short help to open a help window with more help for the option.
Modify the value of the option and hit <CR> on the "set" line to set the new value. For window and buffer specific options, the last accessed window is used to set the option value in, unless this is a help window, in which case the window below help window is used (skipping the option-window).

$HOME

Using "~" is like using "$HOME", but it is only recognized at the start of an option and after a space or comma.

On Unix systems "~user" can be used too. It is replaced by the home directory of user "user". Example:

```
:set path=~mool/include,/usr/include,.
```

On Unix systems the form "${HOME}" can be used too. The name between {} can contain non-id characters then. Note that if you want to use this for the "gf" command, you need to add the '{' and '}' characters to 'isfname'.

NOTE: expanding environment variables and "~/" is only done with the ":set" command, not when assigning a value to an option with ":let".

$HOME-windows

On MS-Windows, if $HOME is not defined as an environment variable, then at runtime Vim will set it to the expansion of $HOMEDRIVE$HOMEPATH. If $HOMEDRIVE is not set then $USERPROFILE is used.

This expanded value is not exported to the environment, this matters when running an external command:

```
:echo system('set | findstr ^HOME=')
```

and

```
:echo luaeval('os.getenv("HOME")')
```

should echo nothing (an empty string) despite exists('$HOME') being true. When setting $HOME to a non-empty string it will be exported to the subprocesses.

Note the maximum length of an expanded option is limited. How much depends on the system, mostly it is something like 256 or 1024 characters.

2. Automatically setting options auto-setting

Besides changing options with the ":set" command, you can set options automatically in various ways:

1. With a config file or a startup argument. You can create an initialization file with :mkvimrc, :mkview and :mksession.
2. autocommands executed when you edit a file.
3. ".nvim.lua" files in the current directory, if 'exrc' is enabled.
4. editorconfig in the current buffer's directory or ancestors.
5. 'modeline' settings found at the beginning or end of the file. See below.

modeline vim: vi: ex: E520
There are two forms of modelines. The first form:
[text{white}]{vi:|vim:|ex:}[white]{options}

```
[text{white}]           empty or any text followed by at least one blank
                        character (<Space> or <Tab>); "ex:" always requires at
                        least one blank character
{vi:|vim:|ex:}          the string "vi:", "vim:" or "ex:"
[white]                 optional white space
{options}               a list of option settings, separated with white space
                        or ':', where each part between ':' is the argument
                        for a ":set" command (can be empty)
```

Examples:
vi:noai:sw=3 ts=6
vim: tw=77

The second form (this is compatible with some versions of Vi):

```
        [text{white}]{vi:|vim:|Vim:|ex:}[white]se[t] {options}:[text]
```

```
[text{white}]           empty or any text followed by at least one blank
                        character (<Space> or <Tab>); "ex:" always requires at
                        least one blank character
{vi:|vim:|Vim:|ex:}     the string "vi:", "vim:", "Vim:" or "ex:"
[white]                 optional white space
se[t]                   the string "set " or "se " (note the space); When
                        "Vim" is used it must be "set".
{options}               a list of options, separated with white space, which
                        is the argument for a ":set" command
:                       a colon
[text]                  any text or empty
```

Examples:

```
/* vim: set ai tw=75: */
/* Vim: set ai tw=75: */
```

The white space before {vi:|vim:|Vim:|ex:} is required. This minimizes the chance that a normal word like "lex:" is caught. There is one exception: "vi:" and "vim:" can also be at the start of the line (for compatibility with version 3.0). Using "ex:" at the start of the line will be ignored (this could be short for "example:").

If the modeline is disabled within a modeline, subsequent modelines will be ignored. This is to allow turning off modeline on a per-file basis. This is useful when a line looks like a modeline but isn't. For example, it would be good to start a YAML file containing strings like "vim:" with
vim: nomodeline
so as to avoid modeline misdetection. Following options on the same line after modeline deactivation, if any, are still evaluated (but you would normally not have any).

modeline-local

The options are set like with ":setlocal": The new value only applies to the buffer and window that contain the file. Although it's possible to set global options from a modeline, this is unusual. If you have two windows open and the files in it set the same global option to a different value, the result depends on which one was opened last.

When editing a file that was already loaded, only the window-local options from the modeline are used. Thus if you manually changed a buffer-local option after opening the file, it won't be changed if you edit the same buffer in another window. But window-local options will be set.

modeline-version

If the modeline is only to be used for some versions of Vim, the version number can be specified where "vim:" or "Vim:" is used:

```
        vim{vers}:      version {vers} or later
        vim<{vers}:     version before {vers}
        vim={vers}:     version {vers}
        vim>{vers}:     version after {vers}
```

{vers} is 700 for Vim 7.0 (hundred times the major version plus minor).
For example, to use a modeline only for Vim 7.0:

```
/* vim700: set foldmethod=marker */
```

To use a modeline for Vim after version 7.2:

```
/* vim>702: set cole=2: */
```

There can be no blanks between "vim" and the ":".
The modeline is ignored if {vers} does not fit in an integer.

The number of lines that are checked can be set with the 'modelines' option.
If 'modeline' is off or 'modelines' is 0 no lines are checked.

Note that for the first form all of the rest of the line is used, thus a line like:

```
/* vi:ts=4: */
```

will give an error message for the trailing "*/". This line is OK:

```
/* vi:set ts=4: */
```

If an error is detected the rest of the line is skipped.

If you want to include a ':' in a set command precede it with a '\'. The backslash in front of the ':' will be removed. Example:

```
/* vi:set fillchars=stl\:^,vert\:\|: */
```

This sets the 'fillchars' option to "stl:^,vert:\|". Only a single backslash before the ':' is removed. Thus to include "\:" you have to specify "\\:".

E992 no-modeline-option

No other commands than "set" are supported, for security reasons (somebody might create a Trojan horse text file with modelines). And not all options can be set. For some options a flag is set, so that when the value is used the sandbox is effective. Some options can only be set from the modeline when 'modelineexpr' is set (the default is off).

Still, there is always a small risk that a modeline causes trouble. E.g., when some joker sets 'textwidth' to 5 all your lines are wrapped unexpectedly. So disable modelines before editing untrusted text. The mail ftplugin does this, for example.

Hint: If you would like to do something else than setting an option, you could define an autocommand that checks the file for a specific string. For example:

```
au BufReadPost * if getline(1) =~ "VAR" | call SetVar() | endif
```

And define a function SetVar() that does something with the line containing "VAR".

3. Options summary option-summary

In the list below all the options are mentioned with their full name and with an abbreviation if there is one. Both forms may be used.

In this document when a boolean option is "set" that means that ":set option" is entered. When an option is "reset", ":set nooption" is used.

Most options are the same in all windows and buffers. There are a few that are specific to how the text is presented in a window. These can be set to a different value in each window. For example the 'list' option can be set in one window and reset in another for the same text, giving both types of view at the same time. There are a few options that are specific to a certain file. These can have a different value for each file or buffer. For example the 'textwidth' option can be 78 for a normal text file and 0 for a C program.

global	one option for all buffers and windows
local to window	each window has its own copy of this option
local to buffer	each buffer has its own copy of this option

When creating a new window the option values from the currently active window are used as a default value for the window-specific options. For the buffer-specific options this depends on the 's' and 'S' flags in the 'cpoptions' option. If 's' is included (which is the default) the values for buffer options are copied from the currently active buffer when a buffer is first entered. If 'S' is present the options are copied each time the buffer is entered, this is almost like having global options. If 's' and 'S' are not present, the options are copied from the currently active buffer when the buffer is created.

Hidden options hidden-options

Not all options are supported in all versions. This depends on the supported features and sometimes on the system. A remark about this is in curly braces below. When an option is not supported, it is called a hidden option. Trying to get the value of a hidden option will not give an error, it will return the default value for that option instead. You can't change the value of a hidden option.

To test if "foo" is a valid option name, use something like this:

```
if exists('&foo')
```

This also returns true for a hidden option. To test if option "foo" is really supported use something like this:

```
if exists('+foo')
```

E355

A jump table for the options with a short description can be found at Q_op.

'allowrevins' 'ari' 'noallowrevins' 'noari'
'allowrevins' 'ari' boolean (default off)
global
Allow CTRL-_ in Insert mode. This is default off, to avoid that users that accidentally type CTRL-_ instead of SHIFT-_ get into reverse Insert mode, and don't know how to get out. See 'revins'.

'ambiwidth' 'ambw'

'ambiwidth' 'ambw' string (default "single")
global

Tells Vim what to do with characters with East Asian Width Class Ambiguous (such as Euro, Registered Sign, Copyright Sign, Greek letters, Cyrillic letters).

There are currently two possible values:

"single": Use the same width as characters in US-ASCII. This is expected by most users.

"double": Use twice the width of ASCII characters.

E834 E835

The value "double" cannot be used if 'listchars' or 'fillchars' contains a character that would be double width. These errors may also be given when calling setcellwidths().

The values are overruled for characters specified with setcellwidths().

There are a number of CJK fonts for which the width of glyphs for those characters are solely based on how many octets they take in legacy/traditional CJK encodings. In those encodings, Euro, Registered sign, Greek/Cyrillic letters are represented by two octets, therefore those fonts have "wide" glyphs for them. This is also true of some line drawing characters used to make tables in text file. Therefore, when a CJK font is used for GUI Vim or Vim is running inside a terminal (emulators) that uses a CJK font (or Vim is run inside an xterm invoked with "-cjkwidth" option.), this option should be set to "double" to match the width perceived by Vim with the width of glyphs in the font. Perhaps it also has to be set to "double" under CJK MS-Windows when the system locale is set to one of CJK locales. See Unicode Standard Annex #11 (https://www.unicode.org/reports/tr11).

'arabic' 'arab' 'noarabic' 'noarab'

'arabic' 'arab' boolean (default off)
local to window

This option can be set to start editing Arabic text.
Setting this option will:

- Set the 'rightleft' option, unless 'termbidi' is set.
- Set the 'arabicshape' option, unless 'termbidi' is set.
- Set the 'keymap' option to "arabic"; in Insert mode CTRL-^ toggles between typing English and Arabic key mapping.
- Set the 'delcombine' option

Resetting this option will:

- Reset the 'rightleft' option.
- Disable the use of 'keymap' (without changing its value).

Note that 'arabicshape' and 'delcombine' are not reset (it is a global option).
Also see l10n-arabic.txt.

'arabicshape' 'arshape' 'noarabicshape' 'noarshape'

'arabicshape' 'arshape' boolean (default on)
global

When on and 'termbidi' is off, the required visual character corrections that need to take place for displaying the Arabic language take effect. Shaping, in essence, gets enabled; the term is a broad one which encompasses:

a) the changing/morphing of characters based on their location within a word (initial, medial, final and stand-alone).
b) the enabling of the ability to compose characters
c) the enabling of the required combining of some characters

When disabled the display shows each character's true stand-alone form.
Arabic is a complex language which requires other settings, for further details see l10n-arabic.txt.

'autochdir' 'acd' 'noautochdir' 'noacd'

'autochdir' 'acd' boolean (default off)
global

When on, Nvim will change its global current-directory to the parent of the current file, whenever you open a file, switch buffers, delete a buffer or open/close a window.

When a buffer has no name it also has no directory, thus the current directory won't change when navigating to it.

Note: When this option is on some plugins may not work.

Alternatively, consider using :bcd, :lcd, or :tcd in an autocmd, to selectively set a local (not global) directory. See project-dir.

'autocomplete' 'ac' 'noautocomplete' 'noac'
'autocomplete' 'ac'	boolean (default off)
global or local to buffer global-local
When on, Vim shows a completion menu as you type, similar to using i_CTRL-N, but triggered automatically. See ins-autocompletion.

'autocompletedelay' 'acl'
'autocompletedelay' 'acl'	number (default 0)
global
Delay in milliseconds before the autocomplete menu appears after typing. If you prefer it not to open too quickly, set this value slightly above your typing speed. See ins-autocompletion.

'autocompletetimeout' 'act'
'autocompletetimeout' 'act'	number (default 80)
global
Initial timeout (in milliseconds) for the decaying time-sliced completion algorithm. Starts at this value, halves for each slower source until a minimum is reached. All sources run, but slower ones are quickly de-prioritized. The default is tuned so the popup menu opens within ~200ms even with multiple slow sources on a slow system. Changing this value is rarely needed. Only 80 or higher is valid. Special case: when 'complete' contains "F" or "o" (function sources), a longer timeout is used, allowing up to ~1s for sources such as LSP servers that may sometimes take longer (e.g., while loading modules). See ins-autocompletion.

'autoindent' 'ai' 'noautoindent' 'noai'
'autoindent' 'ai'	boolean (default on)
local to buffer
Copy indent from current line when starting a new line (typing <CR> in Insert mode or when using the "o" or "O" command). If you do not type anything on the new line except <BS> or CTRL-D and then type <Esc>, CTRL-O or <CR>, the indent is deleted again. Moving the cursor to another line has the same effect, unless the 'I' flag is included in 'cpoptions'.
When autoindent is on, formatting (with the "gq" command or when you reach 'textwidth' in Insert mode) uses the indentation of the first line.
When 'smartindent' or 'cindent' is on the indent is changed in a different way.

'autoread' 'ar' 'noautoread' 'noar'
'autoread' 'ar'	boolean (default on)
global or local to buffer global-local
When a file was changed outside of Nvim, automatically read it again. Skipped if the file was deleted (so you still have the last-available text). If the file appears again, then it is read; you can undo to see the previous contents. timestamp

This is driven (partially) by OS filewatcher events uv_fs_event_t, so buffers are updated immediately (instead of only on focus-change or shell-commands).

If this option has a local value, use this command to switch back to using the global value:

```
set autoread<
```

'autowrite' 'aw' 'noautowrite' 'noaw'
'autowrite' 'aw'	boolean (default off)
global
Write the contents of the file, if it has been modified, on each :next, :rewind, :last, :first, :previous, :tag, :stop, :suspend, :!, :make, :terminal, CTRL-] or CTRL-^ command; and when a :buffer, CTRL-O, CTRL-I, '{A-Z0-9}, or `{A-Z0-9} command switches to another file.
A buffer is not written if it becomes hidden, e.g. when 'bufhidden' is set to "hide" and :next is used.
Note that for some commands the 'autowrite' option is not used, see 'autowriteall' for that.
Some buffers will not be written, specifically when 'buftype' is "nowrite", "nofile", "terminal" or "prompt".
USE WITH CARE: If you make temporary changes to a buffer that you don't want to be saved this option may cause it to be saved anyway. Renaming the buffer with ":file {name}" may help avoid this.

'autowriteall' 'awa' 'noautowriteall' 'noawa'
'autowriteall' 'awa'	boolean (default off)
global
Like 'autowrite', but also used for commands :edit, :enew, :quit, :qall, :exit, :xit, :recover and closing the Vim window.
Setting this option also implies that Vim behaves like 'autowrite' has been set.

'background' 'bg'
'background' 'bg'	string (default "dark")
global
When set to "dark" or "light", adjusts the default color groups for that background type. The TUI or other UI sets this on startup if it can detect the background color, and re-detects it whenever a UI attaches later, unless 'background' was set explicitly. When multiple UIs are attached they share one value, decided by "last wins" (may

not be the most recently-attached UI, since it depends on response speed).

This option does NOT change the background color, it tells Nvim what the "inherited" (terminal/GUI) background looks like. See :hi-normal to set the background color explicitly.

g:colors_name

When a color scheme is loaded (the "g:colors_name" variable is set) changing 'background' will cause the color scheme to be reloaded. If the color scheme adjusts to the value of 'background' this will work. However, if the color scheme sets 'background' itself the effect may be undone. First delete the "g:colors_name" variable when needed.

Historically, this option was set in the vimrc file. Example:

```
if $TERM ==# "xterm"
  set background=dark
endif
```

When this option is changed, the defaults for highlight groups will change. To override those defaults, place ":highlight" commands AFTER setting the 'background' option.

'backspace' 'bs'

'backspace' 'bs' string (default "indent,eol,start")
global

Influences the working of <BS>, <Del>, CTRL-W and CTRL-U in Insert mode. This is a list of items, separated by commas. Each item allows a way to backspace over something:

value	effect
indent	allow backspacing over autoindent
eol	allow backspacing over line breaks (join lines)
start	allow backspacing over the start of insert; CTRL-W and CTRL-U stop once at the start of insert.
nostop	like start, except CTRL-W and CTRL-U do not stop at the start of insert.

When the value is empty, Vi compatible backspacing is used, none of the ways mentioned for the items above are possible.

'backup' 'bk' 'nobackup' 'nobk'

'backup' 'bk' boolean (default off)
global

Make a backup before overwriting a file. Leave it around after the file has been successfully written. If you do not want to keep the backup file, but you do want a backup while the file is being written, reset this option and set the 'writebackup' option (this is the default). If you do not want a backup file at all reset both options (use this if your file system is almost full). See the backup-table for more explanations.
When the 'backupskip' pattern matches, a backup is not made anyway.
When 'patchmode' is set, the backup may be renamed to become the oldest version of a file.

'backupcopy' 'bkc'

'backupcopy' 'bkc' string (default "auto")
global or local to buffer global-local

When writing a file and a backup is made, this option tells how it's done. This is a comma-separated list of words.

The main values are:
"yes" make a copy of the file and overwrite the original one
"no" rename the file and write a new one
"auto" one of the previous, what works best

Extra values that can be combined with the ones above are:
"breaksymlink" always break symlinks when writing
"breakhardlink" always break hardlinks when writing

Making a copy and overwriting the original file:
- Takes extra time to copy the file.

  + When the file has special attributes, is a (hard/symbolic) link or has a resource fork, all this is preserved.
- When the file is a link the backup will have the name of the link,

  not of the real file.

Renaming the file and writing a new one:
+ It's fast.
- Sometimes not all attributes of the file can be copied to the new

  file.
- When the file is a link the new file will not be a link.

The "auto" value is the middle way: When Vim sees that renaming the file is possible without side effects (the attributes can be passed on and the file is not a link) that is used. When problems are expected, a copy will be made.

The "breaksymlink" and "breakhardlink" values can be used in combination with any of "yes", "no" and "auto". When included, they force Vim to always break either symbolic or hard links by doing exactly what the "no" option does, renaming the original file to become the backup and writing a new file in its place. This can be useful for example in source trees where all the files are symbolic or hard links and any changes should stay in the local source tree, not be propagated back to the original source.

crontab

One situation where "no" and "auto" will cause problems: A program that opens a file, invokes Vim to edit that file, and then tests if the open file was changed (through the file descriptor) will check the backup file instead of the newly created file. "crontab -e" is an example, as are several file-watcher daemons like inotify. In that case you probably want to switch this option.

When a copy is made, the original file is truncated and then filled with the new text. This means that protection bits, owner and symbolic links of the original file are unmodified. The backup file, however, is a new file, owned by the user who edited the file. The group of the backup is set to the group of the original file. If this fails, the protection bits for the group are made the same as for others.

When the file is renamed, this is the other way around: The backup has the same attributes of the original file, and the newly written file is owned by the current user. When the file was a (hard/symbolic) link, the new file will not! That's why the "auto" value doesn't rename when the file is a link. The owner and group of the newly written file will be set to the same ones as the original file, but the system may refuse to do this. In that case the "auto" value will again not rename the file.

'backupdir' 'bdir'

'backupdir' 'bdir' string (default ".,$XDG_STATE_HOME/nvim/backup//")
global
Disallowed in modeline. no-modeline-option

List of directories for the backup file, separated with commas.

- The backup file will be created in the first directory in the list where this is possible. If none of the directories exist Nvim will attempt to create the last directory in the list.
- Empty means that no backup file will be created ('patchmode' is impossible!). Writing may fail because of this.
- A directory "." means to put the backup file in the same directory as the edited file.
- A directory starting with "./" (or ".\" for MS-Windows) means to put the backup file relative to where the edited file is. The leading "." is replaced with the path name of the edited file. ("." inside a directory name has no special meaning).
- Spaces after the comma are ignored, other spaces are considered part of the directory name. To have a space at the start of a directory name, precede it with a backslash.
- To include a comma in a directory name precede it with a backslash.
- A directory name may end in an '/'.
- For Unix and Win32, if a directory ends in two path separators "//", the swap file name will be built from the complete path to the file with all path separators changed to percent '%' signs. This will ensure file name uniqueness in the backup directory. On Win32, it is also possible to end with "\\". However, When a separating comma is following, you must use "//", since "\\" will include the comma in the file name. Therefore it is recommended to use '//', instead of '\\'.
- Environment variables are expanded :set_env.
- Careful with '\' characters, type one before a space, type two to get one in the option (see option-backslash), for example:

```
set bdir=c:\\tmp,\ dir\\,with\\,commas,\\\ dir\ with\ spaces
```

See also 'backup' and 'writebackup' options.
If you want to hide your backup files on Unix, consider this value:

```
set backupdir=./.backup,~/.backup,.,/tmp
```

You must create a ".backup" directory in each directory and in your
home directory for this to work properly.
The use of :set+= and :set-= is preferred when adding or removing
directories from the list. This avoids problems when a future version
uses another default.

'backupext' 'bex' E589
'backupext' 'bex'	string (default "~")
global
String which is appended to a file name to make the name of the
backup file. The default is quite unusual, because this avoids
accidentally overwriting existing files with a backup file. You might
prefer using ".bak", but make sure that you don't have files with
".bak" that you want to keep.
Only normal file name characters can be used; /*?[|<> are illegal.

If you like to keep a lot of backups, you could use a BufWritePre
autocommand to change 'backupext' just before writing the file to
include a timestamp.

```
au BufWritePre * let &bex = '-' .. strftime("%Y%b%d%X") .. '~'
```

Use 'backupdir' to put the backup in a different directory.

'backupskip' 'bsk'
'backupskip' 'bsk'	string (default "$TMPDIR/*,$TMP/*,$TEMP/*"
Unix: "/tmp/*,$TMPDIR/*,$TMP/*,$TEMP/*"
Mac: "/private/tmp/*,$TMPDIR/*,$TMP/*,$TEMP/*")
global
A list of file patterns. When one of the patterns matches with the
name of the file which is written, no backup file is created. Both
the specified file name and the full path name of the file are used.
The pattern is used like with :autocmd, see autocmd-pattern.
Watch out for special characters, see option-backslash.
When $TMPDIR, $TMP or $TEMP is not defined, it is not used for the
default value. "/tmp/*" is only used for Unix.

WARNING: Not having a backup file means that when Vim fails to write
your buffer correctly and then, for whatever reason, Vim exits, you
lose both the original file and what you were writing. Only disable
backups if you don't care about losing the file.

Note that environment variables are not expanded. If you want to use
$HOME you must expand it explicitly, e.g.:

```
let &backupskip = escape(expand('$HOME'), '\') .. '/tmp/*'
```

Note that the default also makes sure that "crontab -e" works (when a
backup would be made by renaming the original file crontab won't see
the newly created file). Also see 'backupcopy' and crontab.

'belloff' 'bo'
'belloff' 'bo'	string (default "all")
global
Specifies for which events the bell will not be rung. It is a comma-
separated list of items. For each item that is present, the bell will
be silenced. This is most useful to specify specific events in insert
mode to be silenced.
You can also make it flash by using 'visualbell'.

item	meaning when present
all	All events. backspace When hitting <BS> or <Del> and deleting results in an error.
cursor	Fail to move around using the cursor keys or <PageUp>/<PageDown> in Insert-mode. complete Error occurred when using i_CTRL-X_CTRL-K or i_CTRL-X_CTRL-T.
copy	Cannot copy char from insert mode using i_CTRL-Y or i_CTRL-E.
ctrlg	Unknown Char after <C-G> in Insert mode.
error	Other Error occurred (e.g. try to join last line) (mostly used in Normal-mode or Cmdline-mode).
esc	hitting <Esc> in Normal-mode.
hangul	Ignored.
lang	Calling the beep module for Lua/Mzscheme/TCL.
mess	No output available for g<. showmatch Error occurred for 'showmatch' function. operator Empty region error cpo-E. register Unknown register after <C-R> in Insert-mode.
shell	Bell from shell output :!.
spell	Error happened on spell suggest.
term	Bell from :terminal output. wildmode More matches in cmdline-completion available (depends on the 'wildmode' setting).

This is most useful to fine tune when in Insert mode the bell should
be rung. For Normal mode and Ex commands, the bell is often rung to
indicate that an error occurred. It can be silenced by adding the
"error" keyword.

'binary' 'bin' 'nobinary' 'nobin'
'binary' 'bin'	boolean (default off)

local to buffer
This option should be set before editing a binary file. You can also use the -b Vim argument. When this option is switched on a few options will be changed (also when it already was on):
'textwidth' will be set to 0
'wrapmargin' will be set to 0
'modeline' will be off
'expandtab' will be off
Also, 'fileformat' and 'fileformats' options will not be used, the file is read and written like 'fileformat' was "unix" (a single <NL> separates lines).
The 'fileencoding' and 'fileencodings' options will not be used, the file is read without conversion.
NOTE: When you start editing a(nother) file while the 'bin' option is on, settings from autocommands may change the settings again (e.g., 'textwidth'), causing trouble when editing. You might want to set 'bin' again when the file has been loaded.
The previous values of these options are remembered and restored when 'bin' is switched from on to off. Each buffer has its own set of saved option values.
To edit a file with 'binary' set you can use the ++bin argument. This avoids you have to do ":set bin", which would have effect for all files you edit.
When writing a file the <EOL> for the last line is only written if there was one in the original file (normally Vim appends an <EOL> to the last line if there is none; this would make the file longer). See the 'endofline' option.

'bomb' 'nobomb'
'bomb' boolean (default off)
local to buffer
When writing a file and the following conditions are met, a BOM (Byte Order Mark) is prepended to the file:

- this option is on
- the 'binary' option is off
- 'fileencoding' is "utf-8", "ucs-2", "ucs-4" or one of the little/big endian variants.

Some applications use the BOM to recognize the encoding of the file. Often used for UCS-2 files on MS-Windows. For other applications it causes trouble, for example: "cat file1 file2" makes the BOM of file2 appear halfway through the resulting file. Gcc doesn't accept a BOM.
When Vim reads a file and 'fileencodings' starts with "ucs-bom", a check for the presence of the BOM is done and 'bomb' set accordingly. Unless 'binary' is set, it is removed from the first line, so that you don't see it when editing. When you don't change the options, the BOM will be restored when writing the file.

'breakat' 'brk'
'breakat' 'brk' string (default " ^I!@*-+;:,./?")
global
This option lets you choose which characters might cause a line break if 'linebreak' is on. Only works for ASCII characters.

'breakindent' 'bri' 'nobreakindent' 'nobri'
'breakindent' 'bri' boolean (default off)
local to window
Every wrapped line will continue visually indented (same amount of space as the beginning of that line), thus preserving horizontal blocks of text.

'breakindentopt' 'briopt'
'breakindentopt' 'briopt' string (default "")
local to window
Settings for 'breakindent'. It can consist of the following optional items and must be separated by a comma:

min:{n}	Minimum text width that will be kept after applying 'breakindent', even if the resulting text should normally be narrower. This prevents text indented almost to the right window border occupying lots of vertical space when broken. (default: 20)
shift:{n}	After applying 'breakindent', the wrapped line's beginning will be shifted by the given number of characters. It permits dynamic French paragraph indentation (negative) or emphasizing the line continuation (positive). (default: 0)
sbr	Display the 'showbreak' value before applying the additional indent. (default: off)
list:{n}	Adds an additional indent for lines that match a numbered or bulleted list (using the 'formatlistpat' setting). (default: 0)
list:-1	Uses the width of a match with 'formatlistpat' for indentation.

column:{n} Indent at column {n}. Will overrule the other
sub-options. **Note:** an additional indent may be
added for the 'showbreak' setting.
(default: off)

'bufhidden' 'bh'

'bufhidden' 'bh' string (default "")
local to buffer local-noglobal

This option specifies what happens when a buffer is no longer
displayed in a window:

<empty>	follow the global 'hidden' option
hide	hide the buffer (don't unload it), even if 'hidden' is not set
unload	unload the buffer, even if 'hidden' is set; the :hide command will also unload the buffer
delete	delete the buffer from the buffer list, even if 'hidden' is set; the :hide command will also delete the buffer, making it behave like :bdelete
wipe	wipe the buffer from the buffer list, even if 'hidden' is set; the :hide command will also wipe out the buffer, making it behave like :bwipeout

CAREFUL: when "unload", "delete" or "wipe" is used changes in a buffer
are lost without a warning. Also, these values may break autocommands
that switch between buffers temporarily.
This option is used together with 'buftype' and 'swapfile' to specify
special kinds of buffers. See special-buffers.

'buflisted' 'bl' 'nobuflisted' 'nobl' E85

'buflisted' 'bl' boolean (default on)
local to buffer

When this option is set, the buffer shows up in the buffer list. If
it is reset it is not used for ":bnext", "ls", the Buffers menu, etc.
This option is reset by Vim for buffers that are only used to remember
a file name or marks. Vim sets it when starting to edit a buffer.
But not when moving to a buffer with ":buffer".

'buftype' 'bt' E382

'buftype' 'bt' string (default "")
local to buffer local-noglobal

The value of this option specifies the type of a buffer:

(empty)	Normal buffer.
acwrite	Buffer will always be written with BufWriteCmd.
help	Help buffer (do not set this manually).
nofile	Buffer is not a file, will not be written.
nowrite	Buffer represents a filepath (such as a directory), but will not be written.
prompt	Buffer where only the last section can be edited, for use by plugins. prompt-buffer
quickfix	List of errors :cwindow or locations :lwindow
terminal	terminal-emulator buffer

This option is used together with 'bufhidden' and 'swapfile' to
specify special kinds of buffers. See special-buffers.
Also see win_gettype(), which returns the type of the window.

Be careful with changing this option, it can have many side effects!
One such effect is that Vim will not check the timestamp of the file,
if the file is changed by another program this will not be noticed.

A "quickfix" buffer is only used for the error list and the location
list. This value is set by the :cwindow and :lwindow commands and
you are not supposed to change it.

"nofile" and "nowrite" buffers are similar:

both:	The buffer is not to be written to disk, ":w" doesn't work (":w filename" does work though).
both:	The buffer is never considered to be 'modified'. There is no warning when the changes will be lost, for example when you quit Vim.
both:	A swap file is only created when using too much memory (when 'swapfile' has been reset there is never a swap file).
nofile only:	The buffer name is fixed, it is not handled like a file name. It is not modified in response to a :cd command.
both:	When using ":e bufname" and already editing "bufname" the buffer is made empty and autocommands are triggered as usual for :edit.

E676

"acwrite" implies that the buffer name is not related to a file, like
"nofile", but it will be written. Thus, in contrast to "nofile" and
"nowrite", ":w" does work and a modified buffer can't be abandoned
without saving. For writing there must be matching BufWriteCmd,
FileWriteCmd or FileAppendCmd autocommands.

'busy'

'busy' number (default 0)
local to buffer

Sets a buffer "busy" status. Indicated in the default statusline.
When busy status is larger then 0 busy flag is shown in statusline.
The semantics of "busy" are arbitrary, typically decided by the plugin that owns the buffer.

'casemap' 'cmp'

'casemap' 'cmp'		string (default "internal,keepascii")
			global
	Specifies details about changing the case of letters. It may contain
	these words, separated by a comma:
	internal	Use internal case mapping functions, the current
			locale does not change the case mapping. When
			"internal" is omitted, the towupper() and towlower()
			system library functions are used when available.
	keepascii	For the ASCII characters (0x00 to 0x7f) use the US
			case mapping, the current locale is not effective.
			This probably only matters for Turkish.

'cdhome' 'cdh' 'nocdhome' 'nocdh'

'cdhome' 'cdh'		boolean (default on on Unix, off on Windows)
			global
			Disallowed in modeline. no-modeline-option
	When on, :cd, :tcd, :lcd and :bcd without an argument changes the
	current working directory to the $HOME directory like in Unix.
	When off, those commands just print the current directory name.

'cdpath' 'cd' E344 E346

'cdpath' 'cd'		string (default equivalent to $CDPATH or ",,")
			global
			Disallowed in modeline. no-modeline-option
	This is a list of directories which will be searched when using the
	:cd, :tcd, :lcd and :bcd commands, provided that the directory being
	searched for has a relative path, not an absolute part starting with
	"/", "./" or "../", the 'cdpath' option is not used then.
	The 'cdpath' option's value has the same form and semantics as
	'path'. Also see file-searching.
	The default value is taken from $CDPATH, with a "," prepended to look
	in the current directory first.
	If the default value taken from $CDPATH is not what you want, include
	a modified version of the following command in your vimrc file to
	override it:

```
let &cdpath = ',' .. substitute(substitute($CDPATH, '[, ]', '\\\0', 'g'), ':', ',', 'g')
```

	Environment variables are expanded :set_env.
	(parts of 'cdpath' can be passed to the shell to expand file names).

'cedit'

'cedit'			string (default CTRL-F)
			global
	The key used in Command-line Mode to open the command-line window.
	Only non-printable keys are allowed.
	The key can be specified as a single character, but it is difficult to
	type. The preferred way is to use key-notation (e.g. <Up>, <C-F>) or
	a letter preceded with a caret (e.g. ^F is CTRL-F). Examples:

```
set cedit=^Y
set cedit=<Esc>
```

	Nvi also has this option, but it only uses the first character.
	See cmdwin.

'channel'

'channel'		number (default 0)
			local to buffer
	channel connected to the buffer, or 0 if no channel is connected.
	In a :terminal buffer this is the terminal channel.
	Read-only.

'charconvert' 'ccv' E202 E214 E513

'charconvert' 'ccv'	string (default "")
			global
			Disallowed in modeline. no-modeline-option
	An expression that is used for character encoding conversion. It is
	evaluated when a file that is to be read or has been written has a
	different encoding from what is desired.
	'charconvert' is not used when the internal iconv() function is
	supported and is able to do the conversion. Using iconv() is
	preferred, because it is much faster.
	'charconvert' is not used when reading stdin --, because there is no
	file to convert from. You will have to save the text in a file first.
	The expression must return zero, false or an empty string for success,
	non-zero or true for failure.
	See encoding-names for possible encoding names.
	Additionally, names given in 'fileencodings' and 'fileencoding' are
	used.
	Conversion between "latin1", "unicode", "ucs-2", "ucs-4" and "utf-8"
	is done internally by Vim, 'charconvert' is not used for this.
	Also used for Unicode conversion.
	Example:

```
vim.o.charconvert = function()
  vim.fn.system(('recode %s..%s <%s >%s'):format(
    vim.v.charconvert_from, vim.v.charconvert_to,
    vim.v.fname_in, vim.v.fname_out))
  return vim.v.shell_error
end
```

The related Vim variables are:
```
        v:charconvert_from      name of the current encoding
        v:charconvert_to        name of the desired encoding
        v:fname_in              name of the input file
        v:fname_out             name of the output file
```
Note that v:fname_in and v:fname_out will never be the same.

The advantage of using a function call without arguments is that it is faster, see expr-option-function.

If the 'charconvert' expression starts with s: or <SID>, then it is replaced with the script ID (local-function). Example:

```
set charconvert=s:MyConvert()
set charconvert=<SID>SomeConvert()
```

Otherwise the expression is evaluated in the context of the script where the option was set, thus script-local items are available.

'chistory' 'chi' E1542 E1543

'chistory' 'chi' number (default 10)
global

Number of quickfix lists that should be remembered for the quickfix stack. Must be between 1 and 100. If the option is set to a value that is lower than the amount of entries in the quickfix list stack, entries will be removed starting from the oldest one. If the current quickfix list was removed, then the quickfix list at top of the stack (the most recently created) will be used in its place. For additional info, see quickfix-stack.

'cindent' 'cin' 'nocindent' 'nocin'

'cindent' 'cin' boolean (default off)
local to buffer

Enables automatic C program indenting. See 'cinkeys' to set the keys that trigger reindenting in insert mode and 'cinoptions' to set your preferred indent style.
If 'indentexpr' is not empty, it overrules 'cindent'.
If 'lisp' is not on and both 'indentexpr' and 'equalprg' are empty, the "=" operator indents using this algorithm rather than calling an external program.
See C-indenting.
When you don't like the way 'cindent' works, try the 'smartindent' option or 'indentexpr'.

'cinkeys' 'cink'

'cinkeys' 'cink' string (default "0{,0},0),0],:,0#,!^F,o,O,e")
local to buffer

A list of keys that, when typed in Insert mode, cause reindenting of the current line. Only used if 'cindent' is on and 'indentexpr' is empty.
For the format of this option see cinkeys-format.
See C-indenting.

'cinoptions' 'cino'

'cinoptions' 'cino' string (default "")
local to buffer

The 'cinoptions' affect the way 'cindent' reindents lines in a C program. See cinoptions-values for the values of this option, and C-indenting for info on C indenting in general.

'cinscopedecls' 'cinsd'

'cinscopedecls' 'cinsd' string (default "public,protected,private")
local to buffer

Keywords that are interpreted as a C++ scope declaration by cino-g.
Useful e.g. for working with the Qt framework that defines additional scope declarations "signals", "public slots" and "private slots":

```
set cinscopedecls+=signals,public\ slots,private\ slots
```

'cinwords' 'cinw'

'cinwords' 'cinw' string (default "if,else,while,do,for,switch")
local to buffer

These keywords start an extra indent in the next line when 'smartindent' or 'cindent' is set. For 'cindent' this is only done at an appropriate place (inside {}).
Note that 'ignorecase' isn't used for 'cinwords'. If case doesn't matter, include the keyword both the uppercase and lowercase: "if,If,IF".

'clipboard' 'cb'

'clipboard' 'cb' string (default "")
global

This option is a list of comma-separated names.
These names are recognized:

clipboard-unnamed

unnamed When included, Vim will use the clipboard register "*" for all yank, delete, change and put operations which would normally go to the unnamed register. When a register is explicitly specified, it will always be used regardless of whether "unnamed" is in 'clipboard' or not. The clipboard register can always be explicitly accessed using the "* notation. Also see clipboard.

clipboard-unnamedplus

unnamedplus A variant of the "unnamed" flag which uses the clipboard register "+" (quoteplus) instead of register "*" for all yank, delete, change and put operations which would normally go to the unnamed register. When "unnamed" is also included to the option, yank and delete operations (but not put) will additionally copy the text into register "*". See clipboard.

'cmdheight' 'ch'

'cmdheight' 'ch' number (default 1)
global or local to tabpage
Number of screen lines to use for the command-line. Helps avoiding hit-enter prompts.
The value of this option is stored with the tabpage, so that each tabpage can have a different value.

When 'cmdheight' is zero, there is no command-line unless it is being used. The command-line will cover the last line of the screen when shown.

WARNING: cmdheight=0 is EXPERIMENTAL. Works better with ui2 enabled.

'cmdwinheight' 'cwh'

'cmdwinheight' 'cwh' number (default 7)
global
Number of screen lines to use for the command-line window. cmdwin

'colorcolumn' 'cc'

'colorcolumn' 'cc' string (default "")
local to window
'colorcolumn' is a comma-separated list of screen columns that are highlighted with ColorColumn hl-ColorColumn. Useful to align text. Will make screen redrawing slower.
The screen column can be an absolute number, or a number preceded with '+' or '-', which is added to or subtracted from 'textwidth'.

```
set cc=+1         " highlight column after 'textwidth'
set cc=+1,+2,+3  " highlight three columns after 'textwidth'
hi ColorColumn ctermbg=lightgrey guibg=lightgrey
```

When 'textwidth' is zero then the items with '-' and '+' are not used.
A maximum of 256 columns are highlighted.

'columns' 'co' E594

'columns' 'co' number (default 80 or terminal width)
global
Number of columns of the screen. Normally this is set by the terminal initialization and does not have to be set by hand.
When Vim is running in the GUI or in a resizable window, setting this option will cause the window size to be changed. When you only want to use the size for the GUI, put the command in your ginit.vim file.
When you set this option and Vim is unable to change the physical number of columns of the display, the display may be messed up. For the GUI it is always possible and Vim limits the number of columns to what fits on the screen. You can use this command to get the widest window possible:

```
set columns=9999
```

Minimum value is 12, maximum value is 10000.

'comments' 'com' E524 E525

'comments' 'com' string (default "s1:/*,mb:*,ex:*/,://,b:#,:%,:XCOMM,n:>,fb:-,fb:•")
local to buffer
A comma-separated list of strings that can start a comment line. See format-comments. See option-backslash about using backslashes to insert a space.

'commentstring' 'cms' E537

'commentstring' 'cms' string (default "")
local to buffer
A template for a comment. The "%s" in the value is replaced with the comment text, and should be padded with a space when possible.
Used for commenting and to add markers for folding, see fold-marker.

'complete' 'cpt' E535

'complete' 'cpt' string (default ".,w,b,u,t")
local to buffer
This option controls how completion ins-completion behaves when using CTRL-P, CTRL-N, or ins-autocompletion. It is also used for whole-line completion i_CTRL-X_CTRL-L. It indicates the type of completion and the places to scan. It is a comma-separated list of

flags:
. scan the current buffer ('wrapscan' is ignored)
w scan buffers from other windows
b scan other loaded buffers that are in the buffer list
u scan the unloaded buffers that are in the buffer list
U scan the buffers that are not in the buffer list
k scan the files given with the 'dictionary' option
kspell use the currently active spell checking spell
k{dict} scan the file {dict}. Several "k" flags can be given, patterns are valid too. For example:

```
set cpt=k/usr/dict/*,k~/spanish
```

s scan the files given with the 'thesaurus' option
s{tsr} scan the file {tsr}. Several "s" flags can be given, patterns are valid too.
i scan current and included files
d scan current and included files for defined name or macro i_CTRL-X_CTRL-D
] tag completion
t same as "]"
f scan the buffer names (as opposed to buffer contents)
F{func} call the function {func}. Multiple "F" flags may be specified. Refer to complete-functions for details on how the function is invoked and what it should return. The value can be the name of a function or a Funcref. For Funcref values, spaces must be escaped with a backslash ('\'), and commas with double backslashes ('\\') (see option-backslash). Unlike other sources, functions can provide completions starting from a non-keyword character before the cursor, and their start position for replacing text may differ from other sources. If the Dict returned by the {func} includes {"refresh": "always"}, the function will be invoked again whenever the leading text changes.
If generating matches is potentially slow, call complete_check() periodically to keep Vim responsive. This is especially important for ins-autocompletion.
F equivalent to using "F{func}", where the function is taken from the 'completefunc' option.
o equivalent to using "F{func}", where the function is taken from the 'omnifunc' option.

Unloaded buffers are not loaded, thus their autocmds :autocmd are not executed, this may lead to unexpected completions from some files (gzipped files for example). Unloaded buffers are not scanned for whole-line completion.

CTRL-N, CTRL-P, and ins-autocompletion can be used for any 'iskeyword'-based completion (dictionary i_CTRL-X_CTRL-K, included patterns i_CTRL-X_CTRL-I, tags i_CTRL-X_CTRL-], and normal expansions). With the "F" and "o" flags in 'complete', non-keywords can also be completed.

An optional match limit can be specified for a completion source by appending a caret ("^") followed by a {count} to the source flag. For example: ".^9,w,u,t^5" limits matches from the current buffer to 9 and from tags to 5. Other sources remain unlimited.
Note: The match limit takes effect only during forward completion (CTRL-N) and is ignored during backward completion (CTRL-P).

This option cannot be set in a modeline when 'modelineexpr' is off.

'completefunc' 'cfu'
'completefunc' 'cfu' string (default "")
local to buffer
Disallowed in modeline. no-modeline-option
This option specifies a function to be used for Insert mode completion with CTRL-X CTRL-U. i_CTRL-X_CTRL-U
See complete-functions for an explanation of how the function is invoked and what it should return. The value can be the name of a function, a lambda or a Funcref. See option-value-function for more information.

'completeitemalign' 'cia'
'completeitemalign' 'cia' string (default "abbr,kind,menu")
global
A comma-separated list of strings that controls the alignment and display order of items in the popup menu during Insert mode completion. The supported values are "abbr", "kind", and "menu". These values allow customizing how complete-items are shown in the popup menu. **Note:** must always contain those three values in any order.

'completeopt' 'cot'
'completeopt' 'cot' string (default "menu,popup")
global or local to buffer global-local
A comma-separated list of options for Insert mode completion ins-completion. The supported values are:

fuzzy Enable fuzzy-matching for completion candidates. This allows for more flexible and intuitive matching, where characters can be skipped and matches can be found even if the exact sequence is not typed (disabled for thesaurus completion compl-thesaurus).

longest
When 'autocomplete' is not active, only the longest common prefix of the matches is inserted (disabled for thesaurus completion compl-thesaurus). If the popup menu is displayed, you can use CTRL-L to add more characters. Whether case is ignored depends on the type of completion. For buffer text the 'ignorecase' option applies.

When 'autocomplete' is active and no completion item is selected, the longest common prefix of the matches is inserted after the cursor. The prefix is taken either from all displayed items or only from items in the current buffer. The inserted text is highlighted with hl-PreInsert, and the cursor position does not change (similar to "preinsert"). Press CTRL-Y to accept. See also preinserted().

menu Use a popup menu to show the possible completions. The menu is only shown when there is more than one match. ins-completion-menu

menuone Use the popup menu also when there is only one match. Useful when there is additional information about the match, e.g., what file it comes from.

nearest Matches are listed based on their proximity to the cursor position, unlike the default behavior, which only considers proximity for matches appearing below the cursor. This applies only to matches from the current buffer. No effect if "fuzzy" is present.

noinsert Do not insert any text for a match until the user selects a match from the menu. Only works in combination with "menu" or "menuone". No effect if "longest" is present.

noselect Same as "noinsert", except that no menu item is pre-selected. Takes precedence over "noinsert". Enabled automatically when 'autocomplete' is on, unless "preinsert" is also enabled.

nosort Disable sorting of completion candidates based on fuzzy scores when "fuzzy" is enabled. Candidates will appear in their original order.

popup Show extra information about the currently selected completion in a popup window. Only works in combination with "menu" or "menuone". Overrides "preview".

preinsert
Inserts the text of the first completion candidate beyond the current leader, highlighted with hl-PreInsert.
The cursor does not move.
Requires "fuzzy" to be unset, and either "menuone" in 'completeopt' or 'autocomplete' enabled. When 'autocomplete' is enabled, this does not work if 'ignorecase' is set without 'infercase'.
See also preinserted().

preselect
When one of complete-items has its "preselect" field set (e.g., as indicated by an LSP server), select the first such item in the popupmenu-completion. Takes precedence over "noselect".

Unlike the implicit selection behavior (when "noselect" is not set), this preserves the original sort order and navigates to the preselect item rather than always selecting the first item.

preview Show extra information about the currently selected completion in the preview window. Only works in combination with "menu" or "menuone".

Only "fuzzy", "longest", "popup", "preinsert", "preselect" and "preview" have an effect when 'autocomplete' is enabled.

This option does not apply to cmdline-completion. See 'wildoptions' for that.

'completeslash' 'csl'
'completeslash' 'csl' string (default "")
local to buffer
only modifiable in MS-Windows

When this option is set it overrules 'shellslash' for completion:

- When this option is set to "slash", a forward slash is used for path completion in insert mode. This is useful when editing HTML tag, or Makefile with 'noshellslash' on MS-Windows.
- When this option is set to "backslash", backslash is used. This is useful when editing a batch file with 'shellslash' set on MS-Windows.
- When this option is empty, same character is used as for 'shellslash'.

For Insert mode completion the buffer-local value is used. For
command line completion the global value is used.

'completetimeout' 'cto'
'completetimeout' 'cto' number (default 0)
global
Like 'autocompletetimeout', but applies to i_CTRL-N and i_CTRL-P
completion. Value of 0 disables the timeout; positive values allowed.

'concealcursor' 'cocu'
'concealcursor' 'cocu' string (default "")
local to window
Sets the modes in which text in the cursor line can also be concealed.
When the current mode is listed then concealing happens just like in
other lines.

n	Normal mode
v	Visual mode
i	Insert mode
c	Command line editing, for 'incsearch'

'v' applies to all lines in the Visual area, not only the cursor.
A useful value is "nc". This is used in help files. So long as you
are moving around text is concealed, but when starting to insert text
or selecting a Visual area the concealed text is displayed, so that
you can see what you are doing.
Keep in mind that the cursor position is not always where it's
displayed. E.g., when moving vertically it may change column.

'conceallevel' 'cole'
'conceallevel' 'cole' number (default 0)
local to window
Determine how text with the "conceal" syntax attribute :syn-conceal
is shown:

Value	Effect
0	Text is shown normally
1	Each block of concealed text is replaced with one character. If the syntax item does not have a custom replacement character defined (see :syn-cchar) the character defined in 'listchars' is used. It is highlighted with the "Conceal" highlight group.
2	Concealed text is completely hidden unless it has a custom replacement character defined (see :syn-cchar).
3	Concealed text is completely hidden.

Note: in the cursor line concealed text is not hidden, so that you can
edit and copy the text. This can be changed with the 'concealcursor'
option.

'confirm' 'cf' 'noconfirm' 'nocf'
'confirm' 'cf' boolean (default off)
global
When 'confirm' is on, certain operations that would normally
fail because of unsaved changes to a buffer, e.g. ":q" and ":e",
instead raise a dialog asking if you wish to save the current
file(s). You can still use a ! to unconditionally abandon a buffer.
If 'confirm' is off you can still activate confirmation for one
command only (this is most useful in mappings) with the :confirm
command.
Also see the confirm() function and the 'v' flag in 'guioptions'.

'copyindent' 'ci' 'nocopyindent' 'noci'
'copyindent' 'ci' boolean (default off)
local to buffer
Copy the structure of the existing lines indent when autoindenting a
new line. Normally the new indent is reconstructed by a series of
tabs followed by spaces as required (unless 'expandtab' is enabled,
in which case only spaces are used). Enabling this option makes the
new line copy whatever characters were used for indenting on the
existing line. 'expandtab' has no effect on these characters, a Tab
remains a Tab. If the new indent is greater than on the existing
line, the remaining space is filled in the normal manner.
See 'preserveindent'.

'cpoptions' 'cpo' cpo
'cpoptions' 'cpo' string (default "aABceFs_")
global
A sequence of single character flags. When a character is present
this indicates Vi-compatible behavior. This is used for things where
not being Vi-compatible is mostly or sometimes preferred.
'cpoptions' stands for "compatible-options".
Commas can be added for readability.
To avoid problems with flags that are added in the future, use the
"+=" and "-=" feature of ":set" add-option-flags.

contains	behavior	
cpo-a		
a	When included, a ":read" command with a file name argument will set the alternate file name for the current window. cpo-A	
A	When included, a ":write" command with a file name argument will set the alternate file name for the current window. cpo-b	
b	"|" in a ":map" command is recognized as the end of the map command. The '\' is included in the mapping, the text after the '|' is interpreted as the next command. Use a CTRL-V instead of a backslash to include the '|' in the mapping. Applies to all mapping, abbreviation, menu and autocmd commands. See also map_bar. cpo-B	
B	A backslash has no special meaning in mappings, abbreviations, user commands and the "to" part of the menu commands. Remove this flag to be able to use a backslash like a CTRL-V. For example, the command ":map X \ <Esc>" results in X being mapped to:	
'B' included:	"\^["	(^[is a real <Esc>)
'B' excluded:	"<Esc>" (5 characters) cpo-c	
c	Searching continues at the end of any match at the cursor position, but not further than the start of the next line. When not present searching continues one character from the cursor position. With 'c' "abababababab" only gets three matches when repeating "/abab", without 'c' there are five matches. cpo-C	
C	Do not concatenate sourced lines that start with a backslash. See line-continuation. cpo-d	
d	Using "./" in the 'tags' option doesn't mean to use the tags file relative to the current file, but the tags file in the current directory. cpo-D	
D	Can't use CTRL-K to enter a digraph after Normal mode commands with a character argument, like r, f and t. cpo-e	
e	When executing a register with ":@r", always add a <CR> to the last line, also when the register is not linewise. If this flag is not present, the register is not linewise and the last line does not end in a <CR>, then the last line is put on the command-line and can be edited before hitting <CR>. cpo-E	
E	It is an error when using "y", "d", "c", "g~", "gu" or "gU" on an Empty region. The operators only work when at least one character is to be operated on. Example: This makes "y0" fail in the first column. cpo-f	
f	When included, a ":read" command with a file name argument will set the file name for the current buffer, if the current buffer doesn't have a file name yet. cpo-F	
F	When included, a ":write" command with a file name argument will set the file name for the current buffer, if the current buffer doesn't have a file name yet. Also see cpo-P. cpo-i	
i	When included, interrupting the reading of a file will leave it modified. cpo-I	
I	When moving the cursor up or down just after inserting indent for 'autoindent', do not delete the indent. cpo-J	
J	A sentence has to be followed by two spaces after the '.', '!' or '?'. A <Tab> is not recognized as white space. cpo-K	
K	Don't wait for a key code to complete when it is halfway through a mapping. This breaks mapping <F1><F1> when only part of the second <F1> has been read. It enables cancelling the mapping by typing <F1><Esc>. cpo-l	
l	Backslash in a [] range in a search pattern is taken literally, only "\]", "\^", "\-" and "\\" are special. See /[] 'l' included: "/[\t]" finds <Space>, '\' and 't' 'l' excluded: "/[\t]" finds <Space> and <Tab> cpo-L	
L	When the 'list' option is set, 'wrapmargin', 'textwidth', 'softtabstop' and Virtual Replace mode (see gR) count a <Tab> as two characters, instead of the normal behavior of a <Tab>. cpo-m	
m	When included, a showmatch will always wait half a second. When not included, a showmatch will wait half a second or until a character is typed. 'showmatch' cpo-M	
M	When excluded, "%" matching will take backslashes into account. Thus in "(\( )" and "\((\)" the outer parenthesis match. When included "%" ignores backslashes, which is Vi compatible. cpo-n	
n	When included, the column used for 'number' and 'relativenumber' will also be used for text of wrapped lines. cpo-o	
o	Line offset to search command is not remembered for next search. cpo-O	
O	Don't complain if a file is being overwritten, even when it didn't exist when editing it. This is a protection against a file unexpectedly created by someone else. Vi didn't complain about this. cpo-P	
P	When included, a ":write" command that appends to a file will set the file name for the current buffer, if the current buffer doesn't have a file name yet and the 'F' flag is also included cpo-F. cpo-q	
q	When joining multiple lines leave the cursor at the position where it would be when joining two lines. cpo-r	
r	Redo ("." command) uses "/" to repeat a search command, instead of the actually used search string. cpo-R	
R	Remove marks from filtered lines. Without this flag marks are kept like :keepmarks was used. cpo-s	
s	Set buffer options when entering the buffer for the first time. This is like it is in Vim version 3.0. And it is the default. If not present the options are set when the buffer is created. cpo-S	
S	Set buffer options always when entering a buffer (except 'readonly', 'fileformat', 'filetype' and 'syntax'). This is the (most) Vi compatible setting. The options are set to the values in the current buffer. When you change an option and go to another buffer, the value is copied. Effectively makes the buffer options global to all buffers.	

```
's'     'S'     copy buffer options
no      no      when buffer created
yes     no      when buffer first entered (default)
 X      yes     each time when buffer entered (vi comp.)
```

cpo-t

t	Search pattern for the tag command is remembered for
	"n" command. Otherwise Vim only puts the pattern in
	the history for search pattern, but doesn't change the
	last used search pattern.

cpo-u

u	Undo is Vi compatible. See undo-two-ways.

cpo-v

v	Backspaced characters remain visible on the screen in
	Insert mode. Without this flag the characters are
	erased from the screen right away. With this flag the
	screen newly typed text overwrites backspaced
	characters.

cpo-W

W	Don't overwrite a readonly file. When omitted, ":w!"
	overwrites a readonly file, if possible.

cpo-x

x	<Esc> on the command-line executes the command-line.
	The default in Vim is to abandon the command-line,
	because <Esc> normally aborts a command. c_<Esc>

cpo-X

X	When using a count with "R" the replaced text is
	deleted only once. Also when repeating "R" with "."
	and a count.

cpo-y

y	A yank command can be redone with ".". Think twice if
	you really want to use this, it may break some
	plugins, since most people expect "." to only repeat a
	change.

cpo-Z

Z	When using "w!" while the 'readonly' option is set,
	don't reset 'readonly'.

cpo-!

!	When redoing a filter command, use the last used
	external command, whatever it was. Otherwise the last
	used -filter- command is used.

cpo-$

$	When making a change to one line, don't redisplay the
	line, but put a '$' at the end of the changed text.
	The changed text will be overwritten when you type the
	new text. The line is redisplayed if you type any
	command that moves the cursor from the insertion
	point.

cpo-%

%	Vi-compatible matching is done for the "%" command.
	Does not recognize "#if", "#endif", etc.
	Does not recognize "/*" and "*/".
	Parens inside single and double quotes are also
	counted, causing a string that contains a paren to
	disturb the matching. For example, in a line like
	"if (strcmp("foo(", s))" the first paren does not
	match the last one. When this flag is not included,
	parens inside single and double quotes are treated
	specially. When matching a paren outside of quotes,
	everything inside quotes is ignored. When matching a
	paren inside quotes, it will find the matching one (if
	there is one). This works very well for C programs.
	This flag is also used for other features, such as
	C-indenting.

cpo-+

+	When included, a ":write file" command will reset the
	'modified' flag of the buffer, even though the buffer
	itself may still be different from its file.

cpo->

>	When appending to a register, put a line break before
	the appended text.

cpo-;

;	When using , or ; to repeat the last t search
	and the cursor is right in front of the searched
	character, the cursor won't move. When not included,
	the cursor would skip over it and jump to the
	following occurrence.

cpo-~

~	When included, don't resolve symbolic links when
	changing directory with :cd, :tcd, :lcd, or :bcd.
	This preserves the symbolic link path in buffer names
	and when displaying the current directory. When

excluded (default), symbolic links are resolved to
their target paths.

'cpo-_'

_ When using cw on a word, do not include the
whitespace following the word in the motion.

'cursorbind' 'crb' 'nocursorbind' 'nocrb'
'cursorbind' 'crb' boolean (default off)
local to window
When this option is set, as the cursor in the current
window moves other cursorbound windows (windows that also have
this option set) move their cursors to the corresponding line and
column. This option is useful for viewing the
differences between two versions of a file (see 'diff'); in diff mode,
inserted and deleted lines (though not characters within a line) are
taken into account.

'cursorcolumn' 'cuc' 'nocursorcolumn' 'nocuc'
'cursorcolumn' 'cuc' boolean (default off)
local to window
Highlight the screen column of the cursor with CursorColumn
hl-CursorColumn. Useful to align text. Will make screen redrawing
slower.
If you only want the highlighting in the current window you can use
these autocommands:

```
au WinLeave * set nocursorline nocursorcolumn
au WinEnter * set cursorline cursorcolumn
```

'cursorline' 'cul' 'nocursorline' 'nocul'
'cursorline' 'cul' boolean (default off)
local to window
Highlighting used for the line the cursor is on:

CursorLine the text line hl-CursorLine
CursorLineFold the fold column hl-CursorLineFold
CursorLineSign the sign column hl-CursorLineSign
CursorLineNr the line number, when 'cursorlineopt' contains
"number" hl-CursorLineNr

Useful to easily spot the cursor. Will make screen redrawing slower.
When Visual mode is active the highlighting isn't used to make it
easier to see the selected text.

'cursorlineopt' 'culopt'
'cursorlineopt' 'culopt' string (default "both")
local to window
Comma-separated list of settings for how 'cursorline' is displayed.
Valid values:
"line" Highlight the text line of the cursor with
CursorLine hl-CursorLine.
"screenline" Highlight only the screen line of the cursor with
CursorLine hl-CursorLine.
"number" Highlight the line number of the cursor with
CursorLineNr hl-CursorLineNr.

Special value:
"both" Alias for the values "line,number".

"line" and "screenline" cannot be used together.

'debug'

'debug' string (default "")
global
These values can be used:
msg Error messages that would otherwise be omitted will be given
anyway.
throw Error messages that would otherwise be omitted will be given
anyway and also throw an exception and set v:errmsg.
beep A message will be given when otherwise only a beep would be
produced.
The values can be combined, separated by a comma.
"msg" and "throw" are useful for debugging 'foldexpr', 'formatexpr' or
'indentexpr'.

'define' 'def'
'define' 'def' string (default "")
global or local to buffer global-local
Pattern to be used to find a macro definition. It is a search
pattern, just like for the "/" command. This option is used for the
commands like "[i" and "[d" include-search. The 'isident' option is
used to recognize the defined name after the match:

```
{match with 'define'}{non-ID chars}{defined name}{non-ID char}
```

See option-backslash about inserting backslashes to include a space
or backslash.
For C++ this value would be useful, to include const type declarations:

```
^\(#\s*define\|[a-z]*\s*const\s*[a-z]*\)
```

You can also use "\ze" just before the name and continue the pattern
to check what is following. E.g. for Javascript, if a function is

defined with func_name = function(args):

```
^\s*\ze\i\+\s*=\s*function(
```

If the function is defined with func_name : function() {...:

```
^\s*\ze\i\+\s*[:]\s*(*function\s*(
```

When using the ":set" command, you need to double the backslashes!
To avoid that use :let with a single quote string:

```
let &l:define = '^\s*\ze\k\+\s*=\s*function('
```

'delcombine' 'deco' 'nodelcombine' 'nodeco'
'delcombine' 'deco' boolean (default off)
global
If editing Unicode and this option is set, backspace and Normal mode "x" delete each combining character on its own. When it is off (the default) the character along with its combining characters are deleted.
Note: When 'delcombine' is set "xx" may work differently from "2x"!

This is useful for Arabic, Hebrew and many other languages where one may have combining characters overtop of base characters, and want to remove only the combining ones.

'dictionary' 'dict'
'dictionary' 'dict' string (default "")
global or local to buffer global-local
List of file names, separated by commas, that are used to lookup words for keyword completion commands i_CTRL-X_CTRL-K. Each file should contain a list of words. This can be one word per line, or several words per line, separated by non-keyword characters (white space is preferred). Maximum line length is 510 bytes.

When this option is empty or an entry "spell" is present, and spell checking is enabled, words in the word lists for the currently active 'spelllang' are used. See spell.

To include a comma in a file name precede it with a backslash. Spaces after a comma are ignored, otherwise spaces are included in the file name. See option-backslash about using backslashes.
Environment variables are expanded :set_env.
This has nothing to do with the Dictionary variable type.
Where to find a list of words?

- BSD/macOS include the "/usr/share/dict/words" file.

- Try "apt install spell" to get the "/usr/share/dict/words" file on

apt-managed systems (Debian/Ubuntu).
The use of :set+= and :set-= is preferred when adding or removing directories from the list. This avoids problems when a future version uses another default.
Backticks cannot be used in this option for security reasons.

'diff' 'nodiff'
'diff' boolean (default off)
local to window
Join the current window in the group of windows that shows differences between files. See diff-mode.

'diffanchors' 'dia'
'diffanchors' 'dia' string (default "")
global or local to buffer global-local
List of {address} in each buffer, separated by commas, that are considered anchors when used for diffing. It's valid to specify "$+1" for 1 past the last line. "%" cannot be used for this option. There can be at most 20 anchors set for each buffer.

Each anchor line splits the buffer (the split happens above the anchor), with each part being diff'ed separately before the final result is joined. When more than one {address} are provided, the anchors will be sorted internally by line number. If using buffer local options, each buffer should have the same number of anchors (extra anchors will be ignored). This option is only used when 'diffopt' has "anchor" set. See diff-anchors for more details and examples.

E1550

If some of the {address} do not resolve to a line in each buffer (e.g. a pattern search that does not match anything), none of the anchors will be used.

E1562

Diff anchors can only be used when there are no hidden diff buffers.

'diffexpr' 'dex'
'diffexpr' 'dex' string (default "")
global
Disallowed in modeline. no-modeline-option
Expression which is evaluated to obtain a diff file (either ed-style or unified-style) from two versions of a file. See diff-diffexpr.

'diffopt' 'dip'	string	(default "internal,filler,closeoff,indent-heuristic,inline:char,linematch:40")
		global

Option settings for diff mode. It can consist of the following items.
All are optional. Items must be separated by a comma.

algorithm:{text} Use the specified diff algorithm with the internal diff engine. Currently supported algorithms are:

myers	the default algorithm
minimal	spend extra time to generate the smallest possible diff
patience	patience diff algorithm
histogram	histogram diff algorithm

anchor Anchor specific lines in each buffer to be aligned with each other if 'diffanchors' is set. See diff-anchors.

closeoff When a window is closed where 'diff' is set and there is only one window remaining in the same tabpage with 'diff' set, execute :diffoff in that window. This undoes a :diffsplit command.

context:{n} Use a context of {n} lines between a change and a fold that contains unchanged lines. When omitted a context of six lines is used. When using zero the context is actually one, since folds require a line in between, also for a deleted line. Set it to a very large value (999999) to disable folding completely. See fold-diff.

filler Show filler lines, to keep the text synchronized with a window that has inserted lines at the same position. Mostly useful when windows are side-by-side and 'scrollbind' is set.

foldcolumn:{n} Set the 'foldcolumn' option to {n} when starting diff mode. Without this 2 is used.

followwrap Follow the 'wrap' option and leave as it is.

horizontal Start diff mode with horizontal splits (unless explicitly specified otherwise).

hiddenoff Do not use diff mode for a buffer when it becomes hidden.

iblank Ignore changes where lines are all blank. Adds the "-B" flag to the "diff" command if 'diffexpr' is empty. Check the documentation of the "diff" command for what this does exactly.
NOTE: the diff windows will get out of sync, because no differences between blank lines are taken into account.

icase Ignore changes in case of text. "a" and "A" are considered the same. Adds the "-i" flag to the "diff" command if 'diffexpr' is empty.

indent-heuristic
Use the indent heuristic for the internal diff library.

inline:{text} Highlight inline differences within a change. See view-diffs. Supported values are:

none	Do not perform inline highlighting.
simple	Highlight from first different character to the last one in each line. This is the default if no inline: value is set.
char	Use internal diff to perform a character-wise diff and highlight the difference.
word	Use internal diff to perform a word-wise diff and highlight the difference. Non-alphanumeric multi-byte characters such as emoji and CJK characters are considered individual words. Small gaps of non-word characters (5 bytes or less) between changes are merged into a single highlight block.

internal Use the internal diff library. This is ignored when 'diffexpr' is set. E960
When running out of memory when writing a buffer or the diff is larger than 1 GB this item will be ignored for diffs involving that buffer. Set the 'verbose' option to see when this happens.

iwhite Ignore changes in amount of white space. Adds the "-b" flag to the "diff" command if

'diffexpr' is empty. Check the documentation
of the "diff" command for what this does
exactly. It should ignore adding trailing
white space, but not leading white space.

iwhiteall Ignore all white space changes. Adds the "-w" flag to the "diff" command if 'diffexpr' is empty. Check the documentation of the "diff" command for what this does exactly.

iwhiteeol Ignore white space changes at end of line. Adds the "-Z" flag to the "diff" command if 'diffexpr' is empty. Check the documentation of the "diff" command for what this does exactly.

linematch:{n} Align and mark changes between the most similar lines between the buffers. When the total number of lines in the diff hunk exceeds {n}, the lines will not be aligned because for very large diff hunks there will be a noticeable lag. A reasonable setting is "linematch:60", as this will enable alignment for a 2 buffer diff hunk of 30 lines each, or a 3 buffer diff hunk of 20 lines each. Implicitly sets "filler" when this is set.

vertical Start diff mode with vertical splits (unless explicitly specified otherwise).

Examples:

```
set diffopt=internal,filler,context:4
set diffopt=
set diffopt=internal,filler,foldcolumn:3
set diffopt-=internal  " do NOT use the internal diff parser
```

'digraph' 'dg' 'nodigraph' 'nodg'

'digraph' 'dg' boolean (default off)
global

Enable the entering of digraphs in Insert mode with {char1} <BS> {char2}. See digraphs.

'directory' 'dir'

'directory' 'dir' string (default "$XDG_STATE_HOME/nvim/swap//")
global
Disallowed in modeline. no-modeline-option

List of directory names for the swap file, separated with commas.

Possible items:

- The swap file will be created in the first directory where this is possible. If it is not possible in any directory, but last directory listed in the option does not exist, it is created.
- Empty means that no swap file will be used (recovery is impossible!) and no E303 error will be given.
- A directory "." means to put the swap file in the same directory as the edited file. On Unix, a dot is prepended to the file name, so it doesn't show in a directory listing. On MS-Windows the "hidden" attribute is set and a dot prepended if possible.
- A directory starting with "./" (or ".\" for MS-Windows) means to put the swap file relative to where the edited file is. The leading "." is replaced with the path name of the edited file.
- For Unix and Win32, if a directory ends in two path separators "//", the swap file name will be built from the complete path to the file with all path separators replaced by percent '%' signs (including the colon following the drive letter on Win32). This will ensure file name uniqueness in the preserve directory.
  On Win32, it is also possible to end with "\\". However, When a separating comma is following, you must use "//", since "\\" will include the comma in the file name. Therefore it is recommended to use '//', instead of '\\'.
- Spaces after the comma are ignored, other spaces are considered part of the directory name. To have a space at the start of a directory name, precede it with a backslash.
- To include a comma in a directory name precede it with a backslash.
- A directory name may end in an ':' or '/'.

- Environment variables are expanded :set_env.

- Careful with '\' characters, type one before a space, type two to

  get one in the option (see option-backslash), for example:

```
set dir=c:\\tmp,\ dir\\,with\\,commas,\\\ dir\ with\ spaces
```

Editing the same file twice will result in a warning. Using "/tmp" on is discouraged: if the system crashes you lose the swap file. And others on the computer may be able to see the files.
Use :set+= and :set-= when adding or removing directories from the list, this avoids problems if the Nvim default is changed.

'display' 'dy'

'display' 'dy' string (default "lastline")
global
Change the way text is displayed. This is a comma-separated list of flags:
lastline When included, as much as possible of the last line in a window will be displayed. "@@@" is put in the last columns of the last screen line to indicate the rest of the line is not displayed.
truncate Like "lastline", but "@@@" is displayed in the first column of the last screen line. Overrules "lastline".
uhex Show unprintable characters hexadecimal as <xx> instead of using ^C and ~C.
msgsep Obsolete flag. Allowed but takes no effect. msgsep

When neither "lastline" nor "truncate" is included, a last line that doesn't fit is replaced with "@" lines.

The "@" character can be changed by setting the "lastline" item in 'fillchars'. The character is highlighted with hl-NonText.

'eadirection' 'ead'

'eadirection' 'ead' string (default "both")
global
Tells when the 'equalalways' option applies:
ver vertically, width of windows is not affected
hor horizontally, height of windows is not affected
both width and height of windows is affected

'emoji' 'emo' 'noemoji' 'noemo'

'emoji' 'emo' boolean (default on)
global
When on all Unicode emoji characters are considered to be full width. This excludes "text emoji" characters, which are normally displayed as single width. However, such "text emoji" are treated as full-width emoji if they are followed by the U+FE0F variant selector.

Unfortunately there is no good specification for this and it has been determined on trial-and-error basis. Use the setcellwidths() function to change the behavior.

'encoding' 'enc'

'encoding' 'enc' string (default "utf-8")
global
String-encoding used internally and for RPC communication.
Always UTF-8.

See 'fileencoding' to control file-content encoding.

'endoffile' 'eof' 'noendoffile' 'noeof'

'endoffile' 'eof' boolean (default off)
local to buffer
Indicates that a CTRL-Z character was found at the end of the file when reading it. Normally only happens when 'fileformat' is "dos".
When writing a file and this option is off and the 'binary' option is on, or 'fixeol' option is off, no CTRL-Z will be written at the end of the file.
See eol-and-eof for example settings.

'endofline' 'eol' 'noendofline' 'noeol'

'endofline' 'eol' boolean (default on)
local to buffer
When writing a file and this option is off and the 'binary' option is on, or 'fixeol' option is off, no <EOL> will be written for the last line in the file. This option is automatically set or reset when starting to edit a new file, depending on whether file has an <EOL> for the last line in the file. Normally you don't have to set or reset this option.
When 'binary' is off and 'fixeol' is on the value is not used when writing the file. When 'binary' is on or 'fixeol' is off it is used to remember the presence of a <EOL> for the last line in the file, so that when you write the file the situation from the original file can be kept. But you can change it if you want to.
See eol-and-eof for example settings.

'equalalways' 'ea' 'noequalalways' 'noea'

'equalalways' 'ea' boolean (default on)
global
When on, all the windows are automatically made the same size after

splitting or closing a window. This also happens the moment the option is switched on. When off, splitting a window will reduce the size of the current window and leave the other windows the same. When closing a window the extra lines are given to the window next to it (depending on 'splitbelow' and 'splitright').
When mixing vertically and horizontally split windows, a minimal size is computed and some windows may be larger if there is room. The 'eadirection' option tells in which direction the size is affected. Changing the height and width of a window can be avoided by setting 'winfixheight' and 'winfixwidth', respectively.
If a window size is specified when creating a new window sizes are currently not equalized (it's complicated, but may be implemented in the future).

'equalprg' 'ep'
'equalprg' 'ep' string (default "")
global or local to buffer global-local
Disallowed in modeline. no-modeline-option
External program to use for "=" command. When this option is empty the internal formatting functions are used; either 'lisp', 'cindent' or 'indentexpr'.
Environment variables are expanded :set_env. See option-backslash about including spaces and backslashes.

'errorbells' 'eb' 'noerrorbells' 'noeb'
'errorbells' 'eb' boolean (default off)
global
Ring the bell (beep or screen flash) for error messages. This only makes a difference for error messages, the bell will be used always for a lot of errors without a message (e.g., hitting <Esc> in Normal mode). See 'visualbell' to make the bell behave like a screen flash or do nothing. See 'belloff' to finetune when to ring the bell.

'errorfile' 'ef'
'errorfile' 'ef' string (default "errors.err")
global
Disallowed in modeline. no-modeline-option
Name of the errorfile for the QuickFix mode (see :cf).
When the "-q" command-line argument is used, 'errorfile' is set to the following argument. See -q.
NOT used for the ":make" command. See 'makeef' for that.
Environment variables are expanded :set_env.
See option-backslash about including spaces and backslashes.

'errorformat' 'efm'
'errorformat' 'efm' string (default is very long)
global or local to buffer global-local
Scanf-like description of the format for the lines in the error file (see errorformat).

'eventignore' 'ei'
'eventignore' 'ei' string (default "")
global
A list of autocommand event names, which are to be ignored.
When set to "all" or when "all" is one of the items, all autocommand events are ignored, autocommands will not be executed.
Otherwise this is a comma-separated list of event names. Example:

```
set ei=WinEnter,WinLeave
```

To ignore all but some events, a "-" prefix can be used:

```
:set ei=all,-WinLeave
```

'eventignorewin' 'eiw'
'eventignorewin' 'eiw' string (default "")
local to window
Similar to 'eventignore' but applies to a particular window and its buffers, for which window and buffer related autocommands can be ignored indefinitely without affecting the global 'eventignore'.

Note: The following events are considered to happen outside of a window context and thus cannot be ignored by 'eventignorewin':

ChanClose,
ChanInfo,
ChanOpen,
CmdAtom,
CmdUndefined,
CmdlineChanged,
CmdlineEnter,
CmdlineLeave,
CmdlineLeavePre,
CmdwinEnter,
CmdwinLeave,
ColorScheme,
ColorSchemePre,
CompleteChanged,
CompleteDone,
CompleteDonePre,
DiagnosticChanged,
DiffUpdated,
DirChanged,
DirChangedPre,

ExitPre,
FocusGained,
FocusLost,
FuncUndefined,
LspAttach,
LspDetach,
LspNotify,
LspProgress,
LspRequest,
LspTokenUpdate,
MarkSet,
MenuPopup,
ModeChanged,
OptionSet,
PackChanged,
PackChangedPre,
Progress,
QuickFixCmdPost,
QuickFixCmdPre,
QuitPre,
RemoteReply,
SafeState,
SessionLoadPost,
SessionLoadPre,
SessionWritePost,
SessionWritePre,
ShellCmdPost,
Signal,
SourceCmd,
SourcePost,
SourcePre,
SpellFileMissing,
StdinReadPost,
StdinReadPre,
SwapExists,
Syntax,
TabClosed,
TabClosedPre,
TabEnter,
TabLeave,
TabMoved,
TabNew,
TabNewEntered,
TermClose,
TermEnter,
TermLeave,
TermOpen,
TermRequest,
TermResponse,
UIEnter,
UILeave,
User,
VimEnter,
VimLeave,
VimLeavePre,
VimResized,
VimResume,
VimSuspend,
WinNew,
WinNewPre

'expandtab' 'et' 'noexpandtab' 'noet'
'expandtab' 'et' boolean (default off)
local to buffer

In Insert mode: Use the appropriate number of spaces to insert a <Tab>. Spaces are used in indents with the '>' and '<' commands and when 'autoindent' is on. To insert a real tab when 'expandtab' is on, use CTRL-V<Tab>. See also :retab and ins-expandtab.

'exrc' 'ex' 'noexrc' 'noex' project-config workspace-config
'exrc' 'ex' boolean (default off)
global
Disallowed in modeline. no-modeline-option

Enables project-local configuration. Nvim will execute any .nvim.lua, .nvimrc, or .exrc file found in the current-directory and all parent directories (ordered upwards), if the files are in the trust list. Use :trust to manage trusted files. See also vim.secure.read().

Unset 'exrc' to stop further searching of 'exrc' files in parent directories, similar to editorconfig.root.

To get its own location, a Lua exrc file can use debug.getinfo(). See lua-script-location.

Compare 'exrc' to editorconfig:

- 'exrc' can execute any code; editorconfig only specifies settings.
- 'exrc' is Nvim-specific; editorconfig works in other editors.

To achieve project-local LSP configuration:

1. Enable 'exrc'.

2. Place LSP configs at ".nvim/lsp/*.lua" in your project root.

3. Create ".nvim.lua" in your project root directory with this line:

```
vim.cmd[[set runtimepath+=.nvim]]
```

'fileencoding' 'fenc' E213

'fileencoding' 'fenc' string (default "")
local to buffer

File-content encoding for the current buffer. Conversion is done with iconv() or as specified with 'charconvert'.

When 'fileencoding' is not UTF-8, conversion will be done when writing the file. For reading see below.
When 'fileencoding' is empty, the file will be saved with UTF-8 encoding (no conversion when reading or writing a file).

WARNING: Conversion to a non-Unicode encoding can cause loss of information!

See encoding-names for the possible values. Additionally, values may be specified that can be handled by the converter, see mbyte-conversion.

When reading a file 'fileencoding' will be set from 'fileencodings'. To read a file in a certain encoding it won't work by setting 'fileencoding', use the ++enc argument. One exception: when 'fileencodings' is empty the value of 'fileencoding' is used. For a new file the global value of 'fileencoding' is used.

Prepending "8bit-" and "2byte-" has no meaning here, they are ignored. When the option is set, the value is converted to lowercase. Thus you can set it with uppercase values too. '_' characters are replaced with '-'. If a name is recognized from the list at encoding-names, it is replaced by the standard name. For example "ISO8859-2" becomes "iso-8859-2".

When this option is set, after starting to edit a file, the 'modified' option is set, because the file would be different when written.

Keep in mind that changing 'fenc' from a modeline happens AFTER the text has been read, thus it applies to when the file will be written. If you do set 'fenc' in a modeline, you might want to set 'nomodified' to avoid not being able to ":q".

This option cannot be changed when 'modifiable' is off.

'fileencodings' 'fencs'

'fileencodings' 'fencs' string (default "ucs-bom,utf-8,default,latin1")
global

This is a list of character encodings considered when starting to edit an existing file. When a file is read, Vim tries to use the first mentioned character encoding. If an error is detected, the next one in the list is tried. When an encoding is found that works, 'fileencoding' is set to it. If all fail, 'fileencoding' is set to an empty string, which means that UTF-8 is used.

WARNING: Conversion can cause loss of information! You can use the ++bad argument to specify what is done with characters that can't be converted.

For an empty file or a file with only ASCII characters most encodings will work and the first entry of 'fileencodings' will be used (except "ucs-bom", which requires the BOM to be present). If you prefer another encoding use an BufReadPost autocommand event to test if your preferred encoding is to be used. Example:

```
au BufReadPost * if search('\S', 'w') == 0 |
        \ set fenc=iso-2022-jp | endif
```

This sets 'fileencoding' to "iso-2022-jp" if the file does not contain non-blank characters.
When the ++enc argument is used then the value of 'fileencodings' is not used.
Note that 'fileencodings' is not used for a new file, the global value of 'fileencoding' is used instead. You can set it with:

```
setglobal fenc=iso-8859-2
```

This means that a non-existing file may get a different encoding than an empty file.
The special value "ucs-bom" can be used to check for a Unicode BOM (Byte Order Mark) at the start of the file. It must not be preceded by "utf-8" or another Unicode encoding for this to work properly.
An entry for an 8-bit encoding (e.g., "latin1") should be the last, because Vim cannot detect an error, thus the encoding is always accepted.
The special value "default" can be used for the encoding from the environment. It is useful when your environment uses a non-latin1 encoding, such as Russian.
When a file contains an illegal UTF-8 byte sequence it won't be recognized as "utf-8". You can use the 8g8 command to find the illegal byte sequence.

WRONG VALUES:	WHAT'S WRONG:
latin1,utf-8	"latin1" will always be used
utf-8,ucs-bom,latin1	BOM won't be recognized in a utf-8 file
cp1250,latin1	"cp1250" will always be used

If 'fileencodings' is empty, 'fileencoding' is not modified.
See 'fileencoding' for the possible values.
Setting this option does not have an effect until the next time a file is read.

'fileformat' 'ff'

'fileformat' 'ff' string (default Windows: "dos", Unix: "unix")
local to buffer

This gives the <EOL> of the current buffer, which is used for reading/writing the buffer from/to a file:

dos	<CR><NL>
unix	<NL>
mac	<CR>

When "dos" is used, CTRL-Z at the end of a file is ignored.
See file-formats and file-read.
For the character encoding of the file see 'fileencoding'.
When 'binary' is set, the value of 'fileformat' is ignored, file I/O works like it was set to "unix".
This option is set automatically when starting to edit a file and 'fileformats' is not empty and 'binary' is off.
When this option is set, after starting to edit a file, the 'modified' option is set, because the file would be different when written.
This option cannot be changed when 'modifiable' is off.

'fileformats' 'ffs'

'fileformats' 'ffs' string (default Windows: "dos,unix", Unix: "unix,dos")
global

This gives the end-of-line (<EOL>) formats that will be tried when starting to edit a new buffer and when reading a file into an existing buffer:

- When empty, the format defined with 'fileformat' will be used always. It is not set automatically.
- When set to one name, that format will be used whenever a new buffer is opened. 'fileformat' is set accordingly for that buffer. The 'fileformats' name will be used when a file is read into an existing buffer, no matter what 'fileformat' for that buffer is set to.
- When more than one name is present, separated by commas, automatic <EOL> detection will be done when reading a file. When starting to edit a file, a check is done for the <EOL>:
  1. If all lines end in <CR><NL>, and 'fileformats' includes "dos", 'fileformat' is set to "dos".
  2. If a <NL> is found and 'fileformats' includes "unix", 'fileformat' is set to "unix". Note that when a <NL> is found without a preceding <CR>, "unix" is preferred over "dos".
  3. If 'fileformat' has not yet been set, and if a <CR> is found, and if 'fileformats' includes "mac", 'fileformat' is set to "mac". This means that "mac" is only chosen when:
     "unix" is not present or no <NL> is found in the file, and
     "dos" is not present or no <CR><NL> is found in the file.
     Except: if "unix" was chosen, but there is a <CR> before the first <NL>, and there appear to be more <CR>s than <NL>s in the first few lines, "mac" is used.
  4. If 'fileformat' is still not set, the first name from 'fileformats' is used.

  When reading a file into an existing buffer, the same is done, but this happens like 'fileformat' has been set appropriately for that file only, the option is not changed.
  When 'binary' is set, the value of 'fileformats' is not used.

When Vim starts up with an empty buffer the first item is used. You can overrule this by setting 'fileformat' in your .vimrc.

For systems with a Dos-like <EOL> (<CR><NL>), when reading files that are ":source"ed and for vimrc files, automatic <EOL> detection may be done:

- When 'fileformats' is empty, there is no automatic detection. Dos format will be used.
- When 'fileformats' is set to one or more names, automatic detection is done. This is based on the first <NL> in the file: If there is a

<CR> in front of it, Dos format is used, otherwise Unix format is
used.
Also see file-formats.

'fileignorecase' 'fic' 'nofileignorecase' 'nofic'
'fileignorecase' 'fic' boolean (default on for systems where case in file
names is normally ignored)
global
When set, case is ignored when using file and directory names.

This option is on by default on systems where the filesystem is
traditionally case-insensitive (for example MS-Windows and macOS).
However, Vim cannot determine at runtime whether a particular
filesystem is case-sensitive or case-insensitive.

See 'wildignorecase' for only ignoring case when doing completion.

'filetype' 'ft'
'filetype' 'ft' string (default "")
local to buffer local-noglobal
When this option is set, the FileType autocommand event is triggered.
All autocommands that match with the value of this option will be
executed. Thus the value of 'filetype' is used in place of the file
name.
Otherwise this option does not always reflect the current file type.
This option is normally set when the file type is detected. To enable
this use the ":filetype on" command. :filetype
Setting this option to a different value is most useful in a modeline,
for a file for which the file type is not automatically recognized.
Example, for in an IDL file:

```
/* vim: set filetype=idl : */
```

FileType filetypes
When a dot appears in the value then this separates two filetype
names, it should therefore not be used for a filetype. Example:

```
/* vim: set filetype=c.doxygen : */
```

This will use the "c" filetype first, then the "doxygen" filetype.
This works both for filetype plugins and for syntax files. More than
one dot may appear.
This option is not copied to another buffer, independent of the 's' or
'S' flag in 'cpoptions'.
Only alphanumeric characters, '-' and '_' can be used (and a '.' is
allowed as delimiter when combining different filetypes).

'fillchars' 'fcs'
'fillchars' 'fcs' string (default "")
global or local to window global-local
Characters to fill the statuslines, vertical separators, special
lines in the window and truncated text in the ins-completion-menu.
It is a comma-separated list of items. Each item has a name, a colon
and the value of that item: E1511

item	default	Used for
stl	' '	statusline of the current window
stlnc	' '	statusline of the non-current windows
wbr	' '	window bar
horiz	'─' or '-'	horizontal separators :split
horizup	'┴' or '-'	upwards facing horizontal separator
horizdown	'┬' or '-'	downwards facing horizontal separator
vert	'│' or '|'	vertical separators :vsplit
vertleft	'┤' or '|'	left facing vertical separator
vertright	'├' or '|'	right facing vertical separator
verthoriz	'┼' or '+'	overlapping vertical and horizontal separator
fold	'·' or '-'	filling 'foldtext'
foldopen	'-'	mark the beginning of a fold
foldclose	'+'	show a closed fold
foldsep	'│' or '|'	open fold middle marker
foldinner	none	character to show instead of the numeric foldlevel when it would be repeated in a narrow 'foldcolumn'
diff	'-'	deleted lines of the 'diff' option
msgsep	' '	message separator 'display'
eob	'~'	empty lines at the end of a buffer
lastline	'@'	'display' contains lastline/truncate
trunc	'>'	truncated text in the ins-completion-menu.
truncrl	'<'	same as "trunc" in 'rightleft' mode

Any one that is omitted will fall back to the default.

Note that "horiz", "horizup", "horizdown", "vertleft", "vertright" and
"verthoriz" are only used when 'laststatus' is 3, since only vertical
window separators are used otherwise.

If 'ambiwidth' is "double" then "horiz", "horizup", "horizdown",
"vert", "vertleft", "vertright", "verthoriz", "foldsep" and "fold"
default to single-byte alternatives.

Example:

```
set fillchars=stl:\ ,stlnc:\ ,vert:│,fold:·,diff:-
```

All items support single-byte and multibyte characters. But
double-width characters are not supported. E1512

The highlighting used for these items:

item	highlight group	
stl	StatusLine	hl-StatusLine
stlnc	StatusLineNC	hl-StatusLineNC
wbr	WinBar	hl-WinBar or hl-WinBarNC
horiz	WinSeparator	hl-WinSeparator
horizup	WinSeparator	hl-WinSeparator
horizdown	WinSeparator	hl-WinSeparator
vert	WinSeparator	hl-WinSeparator
vertleft	WinSeparator	hl-WinSeparator
vertright	WinSeparator	hl-WinSeparator
verthoriz	WinSeparator	hl-WinSeparator
fold	Folded	hl-Folded
foldopen	FoldColumn	hl-FoldColumn
foldclose	FoldColumn	hl-FoldColumn
foldsep	FoldColumn	hl-FoldColumn
diff	DiffDelete	hl-DiffDelete
msgsep	MsgSeparator	hl-MsgSeparator
eob	EndOfBuffer	hl-EndOfBuffer
lastline	NonText	hl-NonText
trunc	one of the many Popup menu highlighting groups like hl-PmenuSel	
truncrl	same as "trunc"	

'findfunc' 'ffu' E1514

'findfunc' 'ffu' string (default "")
global or local to buffer global-local
Disallowed in modeline. no-modeline-option

Function that is called to obtain the filename(s) for the :find
command. When this option is empty, the internal file-searching
mechanism is used.

The value can be the name of a function, a lambda or a Funcref.
See option-value-function for more information.

The function is called with two arguments. The first argument is a
String and is the :find command argument. The second argument is
a Boolean and is set to v:true when the function is called to get
a List of command-line completion matches for the :find command.
The function should return a List, which is handled similarly to the
return value of a :command-completion-customlist function.

The function can process all the directories specified in 'path'.

If a match is found, the function should return a List containing
one or more file names. If a match is not found, the function
should return an empty List.

If any errors are encountered during the function invocation, an
empty List is used as the return value.

It is not allowed to change text or jump to another window while
executing the 'findfunc' textlock.

Examples:

```
-- Use vim.fn.glob()
vim.o.findfunc = function(cmdarg, cmdcomplete)
  local pat = cmdcomplete and (cmdarg .. '*') or cmdarg
  return vim.fn.glob(pat, false, true)
end

-- Use the "git ls-files" output
vim.o.findfunc = function(cmdarg, cmdcomplete)
  local fnames = vim.fn.systemlist('git ls-files')
  return vim.tbl_filter(function(v)
    return v:lower():find(cmdarg:lower(), 1, true) ~= nil
  end, fnames)
end
```

'fixendofline' 'fixeol' 'nofixendofline' 'nofixeol'

'fixendofline' 'fixeol' boolean (default on)
local to buffer

When writing a file and this option is on, <EOL> at the end of file
will be restored if missing. Turn this option off if you want to
preserve the situation from the original file.
When the 'binary' option is set the value of this option doesn't
matter.
See the 'endofline' option.
See eol-and-eof for example settings.

'foldclose' 'fcl'

'foldclose' 'fcl' string (default "")
global

When set to "all", a fold is closed when the cursor isn't in it and

its level is higher than 'foldlevel'. Useful if you want folds to automatically close when moving out of them.

'foldcolumn' 'fdc'
'foldcolumn' 'fdc' string (default "0")
local to window
When and how to draw the foldcolumn. Valid values are:
"auto": resize to the minimum amount of folds to display.
"auto:[1-9]": resize to accommodate multiple folds up to the selected level
"0": to disable foldcolumn
"[1-9]": to display a fixed number of columns
See folding.

'foldenable' 'fen' 'nofoldenable' 'nofen'
'foldenable' 'fen' boolean (default on)
local to window
When off, all folds are open. This option can be used to quickly switch between showing all text unfolded and viewing the text with folds (including manually opened or closed folds). It can be toggled with the zi command. The 'foldcolumn' will remain blank when 'foldenable' is off.
This option is set by commands that create a new fold or close a fold.
See folding.

'foldexpr' 'fde'
'foldexpr' 'fde' string (default "0")
local to window
The expression used for when 'foldmethod' is "expr". It is evaluated for each line to obtain its fold level. The context is set to the script where 'foldexpr' was set, script-local items can be accessed.
See fold-expr for the usage.

The expression will be evaluated in the sandbox if set from a modeline, see sandbox-option.
This option can't be set from a modeline when the 'diff' option is on or the 'modelineexpr' option is off.

It is not allowed to change text or jump to another window while evaluating 'foldexpr' textlock.

'foldignore' 'fdi'
'foldignore' 'fdi' string (default "#")
local to window
Used only when 'foldmethod' is "indent". Lines starting with characters in 'foldignore' will get their fold level from surrounding lines. White space is skipped before checking for this character.
The default "#" works well for C programs. See fold-indent.

'foldlevel' 'fdl'
'foldlevel' 'fdl' number (default 0)
local to window
Sets the fold level: Folds with a higher level will be closed.
Setting this option to zero will close all folds. Higher numbers will close fewer folds.
This option is set by commands like zm, zM and zR.
See fold-foldlevel.

'foldlevelstart' 'fdls'
'foldlevelstart' 'fdls' number (default -1)
global
Sets 'foldlevel' when starting to edit another buffer in a window. Useful to always start editing with all folds closed (value zero), some folds closed (one) or no folds closed (99).
This is done before reading any modeline, thus a setting in a modeline overrules this option. Starting to edit a file for diff-mode also ignores this option and closes all folds.
It is also done before BufReadPre autocommands, to allow an autocmd to overrule the 'foldlevel' value for specific files.
When the value is negative, it is not used.

'foldmarker' 'fmr' E536
'foldmarker' 'fmr' string (default "{{{,}}}")
local to window
The start and end marker used when 'foldmethod' is "marker". There must be one comma, which separates the start and end marker. The marker is a literal string (a regular expression would be too slow).
See fold-marker.

'foldmethod' 'fdm'
'foldmethod' 'fdm' string (default "manual")
local to window
The kind of folding used for the current window. Possible values:

fold-manual	manual	Folds are created manually.
fold-indent	indent	Lines with equal indent form a fold.
fold-expr	expr	'foldexpr' gives the fold level of a line.
fold-marker	marker	Markers are used to specify folds.
fold-syntax	syntax	Syntax highlighting items specify folds.
fold-diff	diff	Fold text that is not changed.

'foldminlines' 'fml'
'foldminlines' 'fml' number (default 1)
local to window
Sets the number of screen lines above which a fold can be displayed closed. Also for manually closed folds. With the default value of one a fold can only be closed if it takes up two or more screen lines.

Set to zero to be able to close folds of just one screen line.
Note that this only has an effect on what is displayed. After using
"zc" to close a fold, which is displayed open because it's smaller
than 'foldminlines', a following "zc" may close a containing fold.

'foldnestmax' 'fdn'

'foldnestmax' 'fdn' number (default 20)
local to window
Sets the maximum nesting of folds for the "indent" and "syntax"
methods. This avoids that too many folds will be created. Using more
than 20 doesn't work, because the internal limit is 20.

'foldopen' 'fdo'

'foldopen' 'fdo' string (default "block,hor,mark,percent,quickfix,search,tag,undo")
global
Specifies for which type of commands folds will be opened, if the
command moves the cursor into a closed fold. It is a comma-separated
list of items.
NOTE: When the command is part of a mapping this option is not used.
Add the zv command to the mapping to get the same effect.
(rationale: the mapping may want to control opening folds itself)

item	commands
all	any
block	(, {, [[, [{, etc.
hor	horizontal movements: "l", "w", "fx", etc.
insert	any command in Insert mode
jump	far jumps: "G", "gg", etc.
mark	jumping to a mark: "'m", CTRL-O, etc.
percent	"%"
quickfix	":cn", ":crew", ":make", etc.
search	search for a pattern: "/", "n", "*", "gd", etc. (not for a search pattern in a ":" command) Also for [s and]s.
tag	jumping to a tag: ":ta", CTRL-T, etc.
undo	undo or redo: "u" and CTRL-R When a movement command is used for an operator (e.g., "dl" or "y%") this option is not used. This means the operator will include the whole closed fold. Note that vertical movements are not here, because it would make it very difficult to move onto a closed fold. In insert mode the folds containing the cursor will always be open when text is inserted. To close folds you can re-apply 'foldlevel' with the zx command or set the 'foldclose' option to "all".

'foldtext' 'fdt'

'foldtext' 'fdt' string (default "foldtext()")
local to window
An expression which is used to specify the text displayed for a closed
fold. The context is set to the script where 'foldexpr' was set,
script-local items can be accessed. See fold-foldtext for the
usage.

The expression will be evaluated in the sandbox if set from a
modeline, see sandbox-option.
This option cannot be set in a modeline when 'modelineexpr' is off.

It is not allowed to change text or jump to another window while
evaluating 'foldtext' textlock.

When set to an empty string, foldtext is disabled, and the line
is displayed normally with highlighting and no line wrapping.

'follow' 'nofollow'

'follow' boolean (default off)
local to buffer
Enables multicursor follow-mode: cursor-relative motions performed
by the primary cursor, cascade to all cursors. Toggled by g=
(buffer-local).

'formatexpr' 'fex'

'formatexpr' 'fex' string (default "")
local to buffer
Expression which is evaluated to format a range of lines for the gq
operator or automatic formatting (see 'formatoptions'). When this
option is empty 'formatprg' is used.

The v:lnum variable holds the first line to be formatted.
The v:count variable holds the number of lines to be formatted.
The v:char variable holds the character that is going to be
inserted if the expression is being evaluated due to
automatic formatting. This can be empty. Don't insert
it yet!

Example:

```
set formatexpr=mylang#Format()
```

This will invoke the mylang#Format() function in the
autoload/mylang.vim file in 'runtimepath'. autoload

The advantage of using a function call without arguments is that it is
faster, see expr-option-function.

The expression is also evaluated when 'textwidth' is set and adding
text beyond that limit. This happens under the same conditions as
when internal formatting is used. Make sure the cursor is kept in the

same spot relative to the text then! The mode() function will
return "i" or "R" in this situation.

When the expression evaluates to non-zero Vim will fall back to using
the internal format mechanism.

If the expression starts with s: or <SID>, then it is replaced with
the script ID (local-function). Example:

```
set formatexpr=s:MyFormatExpr()
set formatexpr=<SID>SomeFormatExpr()
```

Otherwise, the expression is evaluated in the context of the script
where the option was set, thus script-local items are available.

The expression will be evaluated in the sandbox when set from a
modeline, see sandbox-option. That stops the option from working,
since changing the buffer text is not allowed.
This option cannot be set in a modeline when 'modelineexpr' is off.

'formatlistpat' 'flp'
'formatlistpat' 'flp'   string (default "^\s*\d\+[\]:.)}\t]\s*")
local to buffer
A pattern that is used to recognize a list header. This is used for
the "n" flag in 'formatoptions'.
The pattern must match exactly the text that will be the indent for
the line below it. You can use /\ze to mark the end of the match
while still checking more characters. There must be a character
following the pattern, when it matches the whole line it is handled
like there is no match.
The default recognizes a number, followed by an optional punctuation
character and white space.

'formatoptions' 'fo'
'formatoptions' 'fo'   string (default "tcqj")
local to buffer
This is a sequence of letters which describes how automatic
formatting is to be done.
See fo-table for possible values and gq for how to format text.
Commas can be inserted for readability.
To avoid problems with flags that are added in the future, use the
"+=" and "-=" feature of ":set" add-option-flags.

'formatprg' 'fp'
'formatprg' 'fp'   string (default "")
global or local to buffer global-local
Disallowed in modeline. no-modeline-option
External program used to format lines with gq. Ignored if
'formatexpr' is set.

If empty, the internal C-indenting function will be used.

The program must take input on stdin and produce output on stdout. The
Unix program "fmt" is such a program. Environment variables are
expanded :set_env. See option-backslash about including spaces
and backslashes.

'fsync' 'fs' 'nofsync' 'nofs'
'fsync' 'fs'   boolean (default on)
global or local to buffer global-local
Disallowed in modeline. no-modeline-option
When on, the OS function fsync() will be called after saving a file
(:write, writefile(), …), swap-file, undo-persistence and shada-file.
This flushes the file to disk, ensuring that it is safely written.
Slow on some systems: writing buffers, quitting Nvim, and other
operations may sometimes take a few seconds.

Files are ALWAYS flushed ('fsync' is ignored) when:

- CursorHold event is triggered
- :preserve is called
- system signals low battery life
- Nvim exits abnormally

This is a global-local option, so it can be set per buffer, for
example when writing to a slow filesystem.

'grepformat' 'gfm'
'grepformat' 'gfm'   string (default "%f:%l:%m,%f:%l%m,%f %l%m")
global or local to buffer global-local
Format to recognize for the ":grep" command output.
This is a scanf-like string that uses the same format as the
'errorformat' option: see errorformat.

Defaults to "%f:%l:%c:%m" if ripgrep ('grepprg') is available.

'grepprg' 'gp'
'grepprg' 'gp'   string (default see below)
global or local to buffer global-local
Disallowed in modeline. no-modeline-option

Program to use for the :grep command.
Note: if you change this then you must also update 'grepformat'.

May contain "%" and "#" characters, are expanded per cmdline-special.
The placeholder "$*" specifies where the arguments will be included.
Environment variables are expanded :set_env. See option-backslash
about including spaces and backslashes.

Special value: When 'grepprg' is set to "internal" the :grep command
works like :vimgrep, :lgrep like :lvimgrep, :grepadd like
:vimgrepadd and :lgrepadd like :lvimgrepadd.

See also :make_makeprg, most of the comments there apply to 'grepprg'.

Defaults to:

- "rg --vimgrep -uu " if ripgrep is available (:checkhealth),
- "grep -HIn $* /dev/null" on Unix,
- "findstr /n $* nul" on Windows.

Ripgrep may perform additional filtering such as using .gitignore rules
and skipping hidden files. This is disabled by default (via "-u") to
more closely match the behaviour of standard grep.
You can make ripgrep match Vim's case handling using the
-i/--ignore-case and -S/--smart-case options. Handle OptionSet to
dynamically update 'grepprg' when e.g. 'ignorecase' is changed.

'guicursor' 'gcr' E545 E546 E548 E549

'guicursor' 'gcr' string (default "n-v-c-sm:block,i-ci-ve:ver25,r-cr-o:hor20,t:block-blinkon500-blinkoff500-TermCursor")
global

Configures the cursor style for each mode. Works in the GUI and many
terminals. See tui-cursor-shape.

To disable cursor-styling, reset the option:

```
set guicursor=
```

To enable mode shapes, "Cursor" highlight, and blinking:

```
set guicursor=n-v-c:block,i-ci-ve:ver25,r-cr:hor20,o:hor50
  \,a:blinkwait700-blinkoff400-blinkon250-Cursor/lCursor
  \,sm:block-blinkwait175-blinkoff150-blinkon175
```

The option is a comma-separated list of parts. Each part consists of a
mode-list and an argument-list:
        mode-list:argument-list,mode-list:argument-list,..
The mode-list is a dash separated list of these modes:
        n   Normal mode
        v   Visual mode
        ve   Visual mode with 'selection' "exclusive" (same as 'v',
                if not specified)
        o   Operator-pending mode
        i   Insert mode
        r   Replace mode
        c   Command-line Normal (append) mode
        ci   Command-line Insert mode
        cr   Command-line Replace mode
        sm   showmatch in Insert mode
        t   Terminal mode
        a   all modes
The argument-list is a dash separated list of these arguments:
        hor{N} horizontal bar, {N} percent of the character height
        ver{N} vertical bar, {N} percent of the character width
        block   block cursor, fills the whole character

- Only one of the above three should be present.
- Default is "block" for each mode.

blinkwait{N} cursor-blinking

blinkon{N}
blinkoff{N}
        blink times for cursor: blinkwait is the delay before
        the cursor starts blinking, blinkon is the time that
        the cursor is shown and blinkoff is the time that the
        cursor is not shown. Times are in msec. When one of
        the numbers is zero, there is no blinking. E.g.:

```
set guicursor=n:blinkon0
```

        Default is "blinkon0" for each mode.
{group-name}
        Highlight group that decides the color and font of the
        cursor.
        In the TUI:

- inverse/reverse and no group-name are interpreted

                        as "host-terminal default cursor colors" which
                        typically means "inverted bg and fg colors".
- ctermfg and guifg are ignored.

                {group-name}/{group-name}
                        Two highlight group names, the first is used when
                        no language mappings are used, the other when they
                        are. language-mapping

        Examples of parts:
           n-c-v:block-nCursor In Normal, Command-line and Visual mode, use a
                                block cursor with colors from the "nCursor"
                                highlight group
           n-v-c-sm:block,i-ci-ve:ver25-Cursor,r-cr-o:hor20
                                In Normal et al. modes, use a block cursor
                                with the default colors defined by the host
                                terminal. In Insert-like modes, use
                                a vertical bar cursor with colors from
                                "Cursor" highlight group. In Replace-like
                                modes, use an underline cursor with
                                default colors.
           i-ci:ver30-iCursor-blinkwait300-blinkon200-blinkoff150
                                In Insert and Command-line Insert mode, use a
                                30% vertical bar cursor with colors from the
                                "iCursor" highlight group. Blink a bit
                                faster.

        The 'a' mode is different. It will set the given argument-list for
        all modes. It does not reset anything to defaults. This can be used
        to do a common setting for all modes. For example, to switch off
        blinking: "a:blinkon0"

        Examples of cursor highlighting:

```
highlight Cursor gui=reverse guifg=NONE guibg=NONE
" Note: gui=reverse overrides colors.
highlight Cursor gui=NONE guifg=bg guibg=fg
```

                                        'guifont' 'gfn' E235 E596
'guifont' 'gfn'   string (default "DejaVu Sans Mono,Courier New,monospace"
                                 Mac: "SF Mono,Menlo,Monaco,Courier New,monospace"
                                 Linux: "Source Code Pro,DejaVu Sans Mono,Courier New,monospace"
                                 MS-Windows: "Cascadia Code,Cascadia Mono,Consolas,Courier New,monospace")
                        global
        This is a list of fonts which will be used for the GUI version of Vim.
        In its simplest form the value is just one font name. When
        the font cannot be found you will get an error message. To try other
        font names a list can be specified, font names separated with commas.
        The first valid font is used.

        Spaces after a comma are ignored. To include a comma in a font name
        precede it with a backslash. Setting an option requires an extra
        backslash before a space and a backslash. See also
        option-backslash. For example:

```
set guifont=Screen15,\ 7x13,font\\,with\\,commas
```

        will make Vim try to use the font "Screen15" first, and if it fails it
        will try to use "7x13" and then "font,with,commas" instead.

        If none of the fonts can be loaded, Vim will keep the current setting.
        If an empty font list is given, Vim will try using other resource
        settings (for X, it will use the Vim.font resource), and finally it
        will try some builtin default which should always be there ("7x13" in
        the case of X). The font names given should be "normal" fonts. Vim
        will try to find the related bold and italic fonts.

        For Win32 and Mac OS:

```
set guifont=*
```

        will bring up a font requester, where you can pick the font you want.

        The font name depends on the GUI used.

        For Mac OSX you can use something like this:

```
set guifont=Monaco:h10
```

                                                                E236

        Note that the fonts must be mono-spaced (all characters have the same
        width).

        To preview a font on X11, you might be able to use the "xfontsel"
        program. The "xlsfonts" program gives a list of all available fonts.

        For the Win32 GUI   E244 E245
- takes these options in the font name:

hXX - height is XX (points, can be floating-point)
wXX - width is XX (points, can be floating-point)
b - bold
i - italic
u - underline
s - strikeout
cXX - character set XX. Valid charsets are: ANSI, ARABIC,
BALTIC, CHINESEBIG5, DEFAULT, EASTEUROPE, GB2312, GREEK,
HANGEUL, HEBREW, JOHAB, MAC, OEM, RUSSIAN, SHIFTJIS,
SYMBOL, THAI, TURKISH, VIETNAMESE ANSI and BALTIC.
Normally you would use "cDEFAULT".

Use a ':' to separate the options.

- A '_' can be used in the place of a space, so you don't need to use

backslashes to escape the spaces.

- Examples:

```
set guifont=courier_new:h12:w5:b:cRUSSIAN
set guifont=Andale_Mono:h7.5:w4.5
```

'guifontwide' 'gfw' E231 E533 E534
'guifontwide' 'gfw' string (default "")
global
Comma-separated list of fonts to be used for double-width characters.
The first font that can be loaded is used.
Note: The size of these fonts must be exactly twice as wide as the one
specified with 'guifont' and the same height.

When 'guifont' has a valid font and 'guifontwide' is empty Vim will
attempt to set 'guifontwide' to a matching double-width font.

'helpfile' 'hf'
'helpfile' 'hf' string (default "$VIMRUNTIME/doc/help.txt")
global
Disallowed in modeline. no-modeline-option
Name of the main help file. All distributed help files should be
placed together in one directory. Additionally, all "doc" directories
in 'runtimepath' will be used.
Environment variables are expanded :set_env. For example:
"$VIMRUNTIME/doc/help.txt". If $VIMRUNTIME is not set, $VIM is also
tried. Also see $VIMRUNTIME and option-backslash about including
spaces and backslashes.

'helpheight' 'hh'
'helpheight' 'hh' number (default 20)
global
Minimal initial height of the help window when it is opened with the
":help" command. The initial height of the help window is half of the
current window, or (when the 'ea' option is on) the same as other
windows. When the height is less than 'helpheight', the height is
set to 'helpheight'. Set to zero to disable.

'helplang' 'hlg'
'helplang' 'hlg' string (default messages language or empty)
global
Comma-separated list of languages. Vim will use the first language
for which the desired help can be found. The English help will always
be used as a last resort. You can add "en" to prefer English over
another language, but that will only find tags that exist in that
language and not in the English help.
Example:

```
set helplang=de,it
```

This will first search German, then Italian and finally English help
files.
When using CTRL-] and ":help!" in a non-English help file Vim will
try to find the tag in the current language before using this option.
See help-translated.

'hidden' 'hid' 'nohidden' 'nohid'
'hidden' 'hid' boolean (default on)
global
When off a buffer is unloaded (including loss of undo information)
when it is abandoned. When on a buffer becomes hidden when it is
abandoned. A buffer displayed in another window does not become
hidden, of course.

Commands that move through the buffer list sometimes hide a buffer
although the 'hidden' option is off when these three are true:

- the buffer is modified

- 'autowrite' is off or writing is not possible

- the '!' flag was used

Also see windows.

To hide a specific buffer use the 'bufhidden' option.
'hidden' is set for one command with ":hide {command}" :hide.

'history' 'hi'

'history' 'hi'		number (default 10000)
			global
A history of ":" commands, and a history of previous search patterns
is remembered. This option decides how many entries may be stored in
each of these histories (see cmdline-editing and 'messagesopt' for
the number of messages to remember).
The maximum value is 10000.

'hlsearch' 'hls' 'nohlsearch' 'nohls'

'hlsearch' 'hls'		boolean (default on)
			global
When there is a previous search pattern, highlight all its matches.
The hl-Search highlight group determines the highlighting for all
matches not under the cursor while the hl-CurSearch highlight group
(if defined) determines the highlighting for the match under the
cursor. If hl-CurSearch is not defined, then hl-Search is used for
both. Note that only the matching text is highlighted, any offsets
are not applied.
See also: 'incsearch' and :match.
When you get bored looking at the highlighted matches, you can turn it
off with :nohlsearch. This does not change the option value, as
soon as you use a search command, the highlighting comes back.
'redrawtime' specifies the maximum time spent on finding matches.
When the search pattern can match an end-of-line, Vim will try to
highlight all of the matched text. However, this depends on where the
search starts. This will be the first line in the window or the first
line below a closed fold. A match in a previous line which is not
drawn may not continue in a newly drawn line.
You can specify whether the highlight status is restored on startup
with the 'h' flag in 'shada' shada-h.

'icon' 'noicon'

'icon'			boolean (default off, on when title can be restored)
			global
When on, the icon text of the window will be set to the value of
'iconstring' (if it is not empty), or to the name of the file
currently being edited. Only the last part of the name is used.
Overridden by the 'iconstring' option.
Only works if the terminal supports setting window icons.

'iconstring'

'iconstring'		string (default "")
			global
When this option is not empty, it will be used for the icon text of
the window. This happens only when the 'icon' option is on.
Only works if the terminal supports setting window icon text
When this option contains printf-style '%' items, they will be
expanded according to the rules used for 'statusline'. See
'titlestring' for example settings.
This option cannot be set in a modeline when 'modelineexpr' is off.

'ignorecase' 'ic' 'noignorecase' 'noic'

'ignorecase' 'ic'	boolean (default off)
			global
Ignore case in search patterns, cmdline-completion, when
searching in the tags file, expr-== and for Insert-mode completion
ins-completion.
Also see 'smartcase' and 'tagcase'.
Can be overruled by using "\c" or "\C" in the pattern, see
/ignorecase.

'iminsert' 'imi'

'iminsert' 'imi'	number (default 0)
			local to buffer
Specifies whether :lmap or an Input Method (IM) is to be used in
Insert mode. Valid values:
		0	:lmap is off and IM is off
		1	:lmap is ON and IM is off
		2	:lmap is off and IM is ON
To always reset the option to zero when leaving Insert mode with <Esc>
this can be used:

```
inoremap <ESC> <ESC>:set iminsert=0<CR>
```

This makes :lmap and IM turn off automatically when leaving Insert
mode.
Note that this option changes when using CTRL-^ in Insert mode
i_CTRL-^.
The value is set to 1 when setting 'keymap' to a valid keymap name.
It is also used for the argument of commands like "r" and "f".

'imsearch' 'ims'

'imsearch' 'ims'	number (default -1)
			local to buffer
Specifies whether :lmap or an Input Method (IM) is to be used when
entering a search pattern. Valid values:
		-1	the value of 'iminsert' is used, makes it look like
			'iminsert' is also used when typing a search pattern
		0	:lmap is off and IM is off

1 :lmap is ON and IM is off
2 :lmap is off and IM is ON
Note that this option changes when using CTRL-^ in Command-line mode
c_CTRL-^.
The value is set to 1 when it is not -1 and setting the 'keymap'
option to a valid keymap name.

'inccommand' 'icm'
'inccommand' 'icm' string (default "nosplit")
global
When nonempty, shows the effects of :substitute, :smagic,
:snomagic and user commands with the :command-preview flag as you
type.

Possible values:
nosplit Shows the effects of a command incrementally in the
buffer.
split Like "nosplit", but also shows partial off-screen
results in a preview window.

If the preview for built-in commands is too slow (exceeds
'redrawtime') then 'inccommand' is automatically disabled until
Command-line-mode is done.

'include' 'inc'
'include' 'inc' string (default "")
global or local to buffer global-local
Pattern to be used to find an include command. It is a search
pattern, just like for the "/" command (See pattern). This option
is used for the commands "[i", "]I", "[d", etc.
Normally the 'isfname' option is used to recognize the file name that
comes after the matched pattern. But if "\zs" appears in the pattern
then the text matched from "\zs" to the end, or until "\ze" if it
appears, is used as the file name. Use this to include characters
that are not in 'isfname', such as a space. You can then use
'includeexpr' to process the matched text.
See option-backslash about including spaces and backslashes.

'includeexpr' 'inex'
'includeexpr' 'inex' string (default "")
local to buffer
Expression to be used to transform the string found with the 'include'
option to a file name. Mostly useful to change "." to "/" for Java:

```
setlocal includeexpr=substitute(v:fname,'\\.','/','g')
```

The "v:fname" variable will be set to the file name that was detected.
Note the double backslash: the :set command first halves them, then
one remains in the value, where "\." matches a dot literally. For
simple character replacements tr() avoids the need for escaping:

```
setlocal includeexpr=tr(v:fname,'.','/')
```

Also used for the gf command if an unmodified file name can't be
found. Allows doing "gf" on the name after an 'include' statement.
Note: Not used for <cfile>.

If the expression starts with s: or <SID>, then it is replaced with
the script ID (local-function). Example:

```
setlocal includeexpr=s:MyIncludeExpr()
setlocal includeexpr=<SID>SomeIncludeExpr()
```

Otherwise, the expression is evaluated in the context of the script
where the option was set, thus script-local items are available.

It is more efficient if the value is just a function call without
arguments, see expr-option-function.

The expression will be evaluated in the sandbox when set from a
modeline, see sandbox-option.
This option cannot be set in a modeline when 'modelineexpr' is off.

It is not allowed to change text or jump to another window while
evaluating 'includeexpr' textlock.

'incsearch' 'is' 'noincsearch' 'nois'
'incsearch' 'is' boolean (default on)
global
While typing a search command, show where the pattern, as it was typed
so far, matches. The matched string is highlighted. If the pattern
is invalid or not found, nothing is shown. The screen will be updated
often, this is only useful on fast terminals.
Note that the match will be shown, but the cursor will return to its
original position when no match is found and when pressing <Esc>. You
still need to finish the search command with <Enter> to move the
cursor to the match.
You can use the CTRL-G and CTRL-T keys to move to the next and
previous match. c_CTRL-G c_CTRL-T
Vim only searches for about half a second. With a complicated
pattern and/or a lot of text the match may not be found. This is to
avoid that Vim hangs while you are typing the pattern.
The hl-IncSearch highlight group determines the highlighting.
When 'hlsearch' is on, all matched strings are highlighted too while
typing a search command. See also: 'hlsearch'.

If you don't want to turn 'hlsearch' on, but want to highlight all matches while searching, you can turn on and off 'hlsearch' with autocmd. Example:

```
augroup vimrc-incsearch-highlight
  autocmd!
  autocmd CmdlineEnter [\/\?] :set hlsearch
  autocmd CmdlineLeave [\/\?] :set nohlsearch
augroup END
```

CTRL-L can be used to add one character from after the current match to the command line. If 'ignorecase' and 'smartcase' are set and the command line has no uppercase characters, the added character is converted to lowercase.
CTRL-R CTRL-W can be used to add the word at the end of the current match, excluding the characters that were already typed (starting from the beginning of the word).

'indentexpr' 'inde'
'indentexpr' 'inde' string (default "")
local to buffer

Expression which is evaluated to obtain the proper indent for a line. It is used when a new line is created, for the = operator and in Insert mode as specified with the 'indentkeys' option. When this option is not empty, it overrules the 'cindent' and 'smartindent' indenting. When 'lisp' is set, this option is only used when 'lispoptions' contains "expr:1". The expression is evaluated with v:lnum set to the line number for which the indent is to be computed. The cursor is also in this line when the expression is evaluated (but it may be moved around).

If the expression starts with s: or <SID>, then it is replaced with the script ID (local-function). Example:

```
set indentexpr=s:MyIndentExpr()
set indentexpr=<SID>SomeIndentExpr()
```

Otherwise, the expression is evaluated in the context of the script where the option was set, thus script-local items are available.

The advantage of using a function call without arguments is that it is faster, see expr-option-function.

The expression must return the number of spaces worth of indent. It can return "-1" to keep the current indent (this means 'autoindent' is used for the indent).
Functions useful for computing the indent are indent(), cindent() and lispindent().
The evaluation of the expression must not have side effects! It must not change the text, jump to another window, etc. Afterwards the cursor position is always restored, thus the cursor may be moved.
Normally this option would be set to call a function:

```
set indentexpr=GetMyIndent()
```

Error messages will be suppressed, unless the 'debug' option contains "msg".
See indent-expression.

The expression will be evaluated in the sandbox when set from a modeline, see sandbox-option.
This option cannot be set in a modeline when 'modelineexpr' is off.

It is not allowed to change text or jump to another window while evaluating 'indentexpr' textlock.

'indentkeys' 'indk'
'indentkeys' 'indk' string (default "0{,0},0),0],:,0#,!^F,o,O,e")
local to buffer

A list of keys that, when typed in Insert mode, cause reindenting of the current line. Only happens if 'indentexpr' isn't empty.
The format is identical to 'cinkeys', see indentkeys-format.
See C-indenting and indent-expression.

'infercase' 'inf' 'noinfercase' 'noinf'
'infercase' 'inf' boolean (default off)
local to buffer

When doing keyword completion in insert mode ins-completion, and 'ignorecase' is also on, the case of the match is adjusted depending on the typed text. If the typed text contains a lowercase letter where the match has an upper case letter, the completed part is made lowercase. If the typed text has no lowercase letters and the match has a lowercase letter where the typed text has an uppercase letter, and there is a letter before it, the completed part is made uppercase. With 'noinfercase' the match is used as-is.

'isfname' 'isf'
'isfname' 'isf' string (default for Windows:
"@,48-57,/,\,.,-,_,+,,,#,$,%,{,},[,],@-@,!,~,="
otherwise: "@,48-57,/,.,-,_,+,,,#,$,%,~,=")
global

The characters specified by this option are included in file names and path names. Filenames are used for commands like "gf", "[i" and in the tags file. It is also used for "\f" in a pattern.
Multi-byte characters 256 and above are always included, only the

characters up to 255 are specified with this option.
For UTF-8 the characters 0xa0 to 0xff are included as well.
Think twice before adding white space to this option. Although a
space may appear inside a file name, the effect will be that Vim
doesn't know where a file name starts or ends when doing completion.
It most likely works better without a space in 'isfname'.

Note that on systems using a backslash as path separator, Vim tries to
do its best to make it work as you would expect. That is a bit
tricky, since Vi originally used the backslash to escape special
characters. Vim will not remove a backslash in front of a normal file
name character on these systems, but it will on Unix and alikes. The
'&' and '^' are not included by default, because these are special for
cmd.exe.

The format of this option is a list of parts, separated with commas.
Each part can be a single character number or a range. A range is two
character numbers with '-' in between. A character number can be a
decimal number between 0 and 255 or the ASCII character itself (does
not work for digits). Example:

```
        "_,-,128-140,#-43"      (include '_' and '-' and the range
                                128 to 140 and '#' to 43)
```

If a part starts with '^', the following character number or range
will be excluded from the option. The option is interpreted from left
to right. Put the excluded character after the range where it is
included. To include '^' itself use it as the last character of the
option or the end of a range. Example:

```
        "^a-z,#,^"      (exclude 'a' to 'z', include '#' and '^')
```

If the character is '@', all characters where isalpha() returns TRUE
are included. Normally these are the characters a to z and A to Z,
plus accented characters. To include '@' itself use "@-@". Examples:

```
        "@,^a-z"        All alphabetic characters, excluding lower
                        case ASCII letters.
        "a-z,A-Z,@-@"   All letters plus the '@' character.
```

A comma can be included by using it where a character number is
expected. Example:

```
        "48-57,,,_"     Digits, comma and underscore.
```

A comma can be excluded by prepending a '^'. Example:

```
        " -~,^,,9"      All characters from space to '~', excluding
                        comma, plus <Tab>.
```

See option-backslash about including spaces and backslashes.

'isident' 'isi'

'isident' 'isi' string (default "@,48-57,_,192-255")
global

The characters given by this option are included in identifiers.
Identifiers are used after a match of the 'define' option. It is also
used for "\i" in a pattern. See 'isfname' for a description of the
format of this option. For '@' only characters up to 255 are used.

Does not affect environment variable names expr-env.

'iskeyword' 'isk'

'iskeyword' 'isk' string (default "@,48-57,_,192-255")
local to buffer

Keywords are used in searching and recognizing with many commands:
"w", "*", "[i", etc. It is also used for "\k" in a pattern. See
'isfname' for a description of the format of this option. For '@'
characters above 255 check the "word" character class (any character
that is categorized as a letter, number or emoji according to the
Unicode general category).

Note that there is a difference between the "\k" character class and
the word motion. The former matches any word character, while the
latter stops at a change of the character class.

For C programs you could use "a-z,A-Z,48-57,_,.,-,>".
For a help file it is set to all non-blank printable characters except
"*", '"' and '|' (so that CTRL-] on a command finds the help for that
command).
When the 'lisp' option is on the '-' character is always included.
This option also influences syntax highlighting, unless the syntax
uses :syn-iskeyword.

'joinspaces' 'js' 'nojoinspaces' 'nojs'

'joinspaces' 'js' boolean (default off)
global

Insert two spaces after a '.', '?' and '!' with a join command.
Otherwise only one space is inserted.

'jumpoptions' 'jop'

'jumpoptions' 'jop' string (default "clean")
global

List of words that change the behavior of the jumplist.

```
  stack         Make the jumplist behave like the tagstack.
                Relative location of entries in the jumplist is
                preserved at the cost of discarding subsequent entries
                when navigating backwards in the jumplist and then
                jumping to a location.  jumplist-stack

  view          When moving through the jumplist, changelist,
                alternate-file, using mark-motions or when popping
                the tagstack try to restore the mark-view in which
                the action occurred.

  clean         Remove unloaded buffers from the jumplist.
```

'keymap' 'kmp' E544

'keymap' 'kmp' string (default "")
local to buffer

Name of a keyboard mapping. See mbyte-keymap.
Setting this option to a valid keymap name has the side effect of setting 'iminsert' to one, so that the keymap becomes effective.
'imsearch' is also set to one, unless it was -1
Only alphanumeric characters, '.', '-' and '_' can be used.

'keymodel' 'km'

'keymodel' 'km' string (default "")
global

List of comma-separated words, which enable special things that keys can do. These values can be used:

- startsel Using a shifted special key starts selection (either Select mode or Visual mode, depending on "key" being present in 'selectmode').
- stopsel Using a not-shifted special key stops selection.

Special keys in this context are the cursor keys, <End>, <Home>, <PageUp> and <PageDown>.

'keywordprg' 'kp'

'keywordprg' 'kp' string (default ":Man", Windows: ":help")
global or local to buffer global-local
Disallowed in modeline. no-modeline-option

Program to use for the K command. Environment variables are expanded :set_env.

Special cases:

- ":help" opens the word at cursor using :help. (Note that previously setting the global option to the empty value did this, which is now deprecated.)
- ":help!" performs :help! (DWIM) on the WORD at cursor.
- If the value starts with ":", it is invoked as an Ex command and [count] is passed as the first argument, if present.
- If "man" or "man -s", [count] is the manpage section number.

See option-backslash about including spaces and backslashes.

Example:

```
set keywordprg=:help!
set keywordprg=man\ -s
set keywordprg=:Man
```

'langmap' 'lmap' E357 E358

'langmap' 'lmap' string (default "")
global
Disallowed in modeline. no-modeline-option

This option allows switching your keyboard into a special language mode. When you are typing text in Insert mode the characters are inserted directly. When in Normal mode the 'langmap' option takes care of translating these special characters to the original meaning of the key. This means you don't have to change the keyboard mode to be able to execute Normal mode commands.
This is the opposite of the 'keymap' option, where characters are mapped in Insert mode.
Also consider setting 'langremap' to off, to prevent 'langmap' from applying to characters resulting from a mapping.

Example (for Greek, in UTF-8): greek

```
set
langmap=AA,BB,ΨC,ΔD,EE,ΦF,ΓG,HH,II,ΞJ,KK,ΛL,MM,NN,OO,ΠP,QQ,PR,ΣS,TT,ΘU,ΩV,WW,XX,YY,ZZ,αa,βb,ψc,δd,εe,φf,γg,ηh,ιi,ξj,κk,λl,μm,νn,οο,πp,qq,ρr
```

Example (exchanges meaning of z and y for commands):

```
set langmap=zy,yz,ZY,YZ
```

The 'langmap' option is a list of parts, separated with commas. Each part can be in one of two forms:

1. A list of pairs. Each pair is a "from" character immediately followed by the "to" character. Examples: "aA", "aAbBcC".
2. A list of "from" characters, a semicolon and a list of "to" characters. Example: "abc;ABC"
   Example: "aA,fgh;FGH,cCdDeE"
   Special characters need to be preceded with a backslash. These are ";", ',', '"', '|' and backslash itself.

This will allow you to activate vim actions without having to switch back and forth between the languages. Your language characters will be understood as normal vim English characters (according to the

langmap mappings) in the following cases:
 o Normal/Visual mode (commands, buffer/register names, user mappings)
 o Insert/Replace Mode: Register names after CTRL-R
 o Insert/Replace Mode: Mappings
Characters entered in Command-line mode will NOT be affected by
this option. Note that this option can be changed at any time
allowing to switch between mappings for different languages/encodings.
Use a mapping to avoid having to type it each time!

'langmenu' 'lm'
'langmenu' 'lm' string (default "")
global
Language to use for menu translation. Tells which file is loaded
from the "lang" directory in 'runtimepath':

```
"lang/menu_" .. &langmenu .. ".vim"
```

(without the spaces). For example, to always use the Dutch menus, no
matter what $LANG is set to:

```
set langmenu=nl_NL.ISO_8859-1
```

When 'langmenu' is empty, v:lang is used.
Only normal file name characters can be used, /*?[|<> are illegal.
If your $LANG is set to a non-English language but you do want to use
the English menus:

```
set langmenu=none
```

This option must be set before loading menus, switching on filetype
detection or syntax highlighting. Once the menus are defined setting
this option has no effect. But you could do this:

```
source $VIMRUNTIME/delmenu.vim
set langmenu=de_DE.ISO_8859-1
source $VIMRUNTIME/menu.vim
```

Warning: This deletes all menus that you defined yourself!

'langremap' 'lrm' 'nolangremap' 'nolrm'
'langremap' 'lrm' boolean (default off)
global
When off, setting 'langmap' does not apply to characters resulting
from a mapping. If setting 'langmap' disables some of your mappings,
make sure this option is off.

'laststatus' 'ls'
'laststatus' 'ls' number (default 2)
global
The value of this option influences when the last window will have a
status line:
0: never
1: only if there are at least two windows
2: always
3: always and ONLY the last window

Here "last window" means the last window in a column, i.e. the bottom-
most one, just above the command line.

The screen looks nicer with a status line if you have several
windows, but it takes another screen line. status-line

'lazyredraw' 'lz' 'nolazyredraw' 'nolz'
'lazyredraw' 'lz' boolean (default off)
global
When this option is set, the screen will not be redrawn while
executing macros, registers and other commands that have not been
typed. Also, updating the window title is postponed. To force an
update use :redraw.
This may occasionally cause display errors. It is only meant to be
set temporarily when performing an operation where redrawing may cause
flickering or cause a slowdown.

'lhistory' 'lhi'
'lhistory' 'lhi' number (default 10)
local to window
Like 'chistory', but for the location list stack associated with a
window. If the option is changed in either the location list window
itself or the window that is associated with the location list stack,
the new value will also be applied to the other one. This means this
value will always be the same for a given location list window and its
corresponding window. See quickfix-stack for additional info.

'linebreak' 'lbr' 'nolinebreak' 'nolbr'
'linebreak' 'lbr' boolean (default off)
local to window
If on, Vim will wrap long lines at a character in 'breakat' rather
than at the last character that fits on the screen. Unlike
'wrapmargin' and 'textwidth', this does not insert <EOL>s in the file,
it only affects the way the file is displayed, not its contents.
If 'breakindent' is set, line is visually indented. Then, the value
of 'showbreak' is used to put in front of wrapped lines. This option
is not used when the 'wrap' option is off.

Note that <Tab> characters after an <EOL> are mostly not displayed
with the right amount of white space.

'lines' E593

'lines' number (default 24 or terminal height)
global
Number of lines of the Vim window.
Normally you don't need to set this. It is done automatically by the
terminal initialization code.
When Vim is running in the GUI or in a resizable window, setting this
option will cause the window size to be changed. When you only want
to use the size for the GUI, put the command in your gvimrc file.
Vim limits the number of lines to what fits on the screen. You can
use this command to get the tallest window possible:

```
set lines=999
```

Minimum value is 2, maximum value is 1000.

'linespace' 'lsp'

'linespace' 'lsp' number (default 0)
global
only in the GUI
Number of pixel lines inserted between characters. Useful if the font
uses the full character cell height, making lines touch each other.
When non-zero there is room for underlining.
With some fonts there can be too much room between lines (to have
space for ascents and descents). Then it makes sense to set
'linespace' to a negative value. This may cause display problems
though!

'lisp' 'nolisp'

'lisp' boolean (default off)
local to buffer
Lisp mode: When <Enter> is typed in insert mode set the indent for
the next line to Lisp standards (well, sort of). Also happens with
"cc" or "S". 'autoindent' must also be on for this to work.
Also see 'lispwords'.
The '-' character is included in keyword characters. Redefines the
"=" operator to use this same indentation algorithm rather than
calling an external program if 'equalprg' is empty.

'lispoptions' 'lop'

'lispoptions' 'lop' string (default "")
local to buffer
Comma-separated list of items that influence the Lisp indenting when
enabled with the 'lisp' option. Currently only one item is supported:
expr:1 use 'indentexpr' for Lisp indenting when it is set
expr:0 do not use 'indentexpr' for Lisp indenting (default)
Note that when using 'indentexpr' the = operator indents all the
lines, otherwise the first line is not indented (Vi-compatible).

'lispwords' 'lw'

'lispwords' 'lw' string (default is very long)
global or local to buffer global-local
Comma-separated list of words that influence the Lisp indenting when
enabled with the 'lisp' option.

'list' 'nolist'

'list' boolean (default off)
local to window
List mode: By default, show tabs as ">", trailing spaces as "-", and
non-breakable space characters as "+". Useful to see the difference
between tabs and spaces and for trailing blanks. Further changed by
the 'listchars' option.

When 'listchars' does not contain "tab" field, tabs are shown as "^I"
or "<09>", like how unprintable characters are displayed.

The cursor is displayed at the start of the space a Tab character
occupies, not at the end as usual in Normal mode. To get this cursor
position while displaying Tabs with spaces, use:

```
let &list = v:true | let &lcs = 'tab:  '
```

Note that list mode will also affect formatting (set with 'textwidth'
or 'wrapmargin') when 'cpoptions' includes 'L'. See 'listchars' for
changing the way tabs are displayed.

'listchars' 'lcs'

'listchars' 'lcs' string (default "tab:> ,trail:-,nbsp:+")
global or local to window global-local
Strings to use in 'list' mode and for the :list command. It is a
comma-separated list of string settings. E1511

lcs-eol

eol:c Character to show at the end of each line. When
omitted, there is no extra character at the end of the
line.

lcs-tab

tab:xy[z] Two or three characters to be used to show a tab.
The third character is optional.

tab:xy The 'x' is always used, then 'y' as many times as will fit. Thus "tab:>-" displays:

```
>
>-
>--
etc.
```

tab:xyz The 'z' is always used, then 'x' is prepended, and then 'y' is used as many times as will fit. Thus "tab:<->" displays:

```
>
<>
<->
<-->
etc.
```

When "tab:" is omitted, a tab is shown as ^I.

lcs-space

space:c Character to show for a space. When omitted, spaces are left blank.

lcs-multispace

multispace:c...
One or more characters to use cyclically to show for multiple consecutive spaces. Overrides the "space" setting, except for single spaces. When omitted, the "space" setting is used. For example, :set listchars=multispace:---+ shows ten consecutive spaces as:

```
---+---+--
```

lcs-lead

lead:c Character to show for leading spaces. When omitted, leading spaces are blank. Overrides the "space" and "multispace" settings for leading spaces. You can combine it with "tab:", for example:

```
set listchars+=tab:>-,lead:.
```

lcs-leadmultispace indent-guides

leadmultispace:c...
Like the lcs-multispace value, but for leading spaces only. Also overrides lcs-lead for leading multiple spaces.
:set listchars=leadmultispace:---+ shows ten consecutive leading spaces as:

```
---+---+--XXX
```

Where "XXX" denotes the first non-blank characters in the line.

Combined with lcs-leadtab, this can be used to show "indentation guides" (vertical lines).
For example, with 'shiftwidth' 2:

```
set list listchars=leadtab:\ \ |,tab:\ \ |,leadmultispace:\ \ |
```

For richer rendering (per-level colors, treesitter-aware scopes, etc.) use a third-party plugin.

lcs-leadtab

leadtab:xy[z]
Like lcs-tab, but only for leading tabs. When omitted, the "tab" setting is used for leading tabs. lcs-tab must also be set for this to work. E1572
You can combine it with "tab:", for example:

```
let &listchars = 'tab:>-,leadtab:. '
```

This shows leading tabs as periods(.) and other tabs as ">--".

lcs-trail

trail:c Character to show for trailing spaces. When omitted, trailing spaces are blank. Overrides the "space" and "multispace" settings for trailing spaces.

lcs-extends

extends:c Character to show in the last column, when 'wrap' is off and the line continues beyond the right of the screen.

lcs-precedes

precedes:c Character to show in the first visible column of the

physical line, when there is text preceding the
character visible in the first column.

lcs-conceal

conceal:c Character to show in place of concealed text, when
'conceallevel' is set to 1. A space when omitted.

lcs-nbsp

nbsp:c Character to show for a non-breakable space character
(0xA0 (160 decimal) and U+202F). Left blank when
omitted.

The characters ':' and ',' should not be used. UTF-8 characters can
be used. All characters must be single width. E1512

Each character can be specified as hex:

```
set listchars=eol:\\x24
set listchars=eol:\\u21b5
set listchars=eol:\\U000021b5
```

Note that a double backslash is used. The number of hex characters
must be exactly 2 for \\x, 4 for \\u and 8 for \\U.

Examples:

```
set lcs=tab:>-,trail:-
set lcs=tab:>-,eol:<,nbsp:%
set lcs=extends:>,precedes:<
```

hl-NonText highlighting will be used for "eol", "extends" and
"precedes". hl-Whitespace for "nbsp", "space", "tab", "multispace",
"lead" and "trail".

'loadplugins' 'lpl' 'noloadplugins' 'nolpl'
'loadplugins' 'lpl' boolean (default on)
global
When on the plugin scripts are loaded when starting up load-plugins.
This option can be reset in your vimrc file to disable the loading
of plugins.
Note that using the "-u NONE" and "--noplugin" command line arguments
reset this option. -u --noplugin

'makeef' 'mef'
'makeef' 'mef' string (default "")
global
Disallowed in modeline. no-modeline-option
Name of the errorfile for the :make command (see :make_makeprg)
and the :grep command.
When it is empty, an internally generated temp file will be used.
When "##" is included, it is replaced by a number to make the name
unique. This makes sure that the ":make" command doesn't overwrite an
existing file.
NOT used for the ":cf" command. See 'errorfile' for that.
Environment variables are expanded :set_env.
See option-backslash about including spaces and backslashes.

'makeencoding' 'menc'
'makeencoding' 'menc' string (default "")
global or local to buffer global-local
Encoding used for reading the output of external commands. When
empty, encoding is not converted.
This is used for :make, :lmake, :grep, :lgrep, :grepadd,
:lgrepadd, :cfile, :cgetfile, :caddfile, :lfile, :lgetfile,
and :laddfile.

This would be mostly useful when you use MS-Windows. If iconv is
enabled, setting 'makeencoding' to "char" has the same effect as
setting to the system locale encoding. Example:

```
set makeencoding=char   " system locale is used
```

'makeprg' 'mp'
'makeprg' 'mp' string (default "make")
global or local to buffer global-local
Disallowed in modeline. no-modeline-option
Program to use for the ":make" command. See :make_makeprg.
This option may contain '%' and '#' characters (see :_% and :_#),
which are expanded to the current and alternate file name. Use ::S
to escape file names in case they contain special characters.
Environment variables are expanded :set_env. See option-backslash
about including spaces and backslashes.
Note that a '|' must be escaped twice: once for ":set" and once for
the interpretation of a command. When you use a filter called
"myfilter" do it like this:

```
set makeprg=gmake\ \\\|\ myfilter
```

The placeholder "$*" can be given (even multiple times) to specify
where the arguments will be included, for example:

```
set makeprg=latex\ \\\\nonstopmode\ \\\\input\\{$*}
```

'matchpairs' 'mps'

'matchpairs' 'mps'	string	(default "(:),{:},[:]")
		local to buffer
	Characters that form pairs. The % command jumps from one to the
	other.
	Only character pairs are allowed that are different, thus you cannot
	jump between two double quotes.
	The characters must be separated by a colon.
	The pairs must be separated by a comma. Example for including '<' and
	'>' (for HTML):

```
set mps+=<:>
```

	A more exotic example, to jump between the '=' and ';' in an
	assignment, useful for languages like C and Java:

```
au FileType c,cpp,java set mps+==:;
```

	For a more advanced way of using "%", see the matchit.vim plugin in
	the $VIMRUNTIME/plugin directory. add-local-help

'matchtime' 'mat'

'matchtime' 'mat'	number	(default 5)
		global
	Tenths of a second to show the matching paren, when 'showmatch' is
	set. Note that this is not in milliseconds, like other options that
	set a time. This is to be compatible with Nvi.

'maxfuncdepth' 'mfd'

'maxfuncdepth' 'mfd'	number	(default 100)
		global
	Maximum depth of function calls for user functions. This normally
	catches endless recursion. When using a recursive function with
	more depth, set 'maxfuncdepth' to a bigger number. But this will use
	more memory, there is the danger of failing when memory is exhausted.
	Increasing this limit above 200 also changes the maximum for Ex
	command recursion, see E169.
	See also :function.
	Also used for maximum depth of callback functions.

'maxmapdepth' 'mmd' E223

'maxmapdepth' 'mmd'	number	(default 1000)
		global
	Maximum number of times a mapping is done without resulting in a
	character to be used. This normally catches endless mappings, like
	":map x y" with ":map y x". It still does not catch ":map g wg",
	because the 'w' is used before the next mapping is done. See also
	key-mapping.

'maxmempattern' 'mmp'

'maxmempattern' 'mmp'	number	(default 1000)
		global
	Maximum amount of memory (in Kbyte) to use for pattern matching.
	The maximum value is about 2000000. Use this to work without a limit.
E363

	When Vim runs into the limit it gives an error message and mostly
	behaves like CTRL-C was typed.
	Running into the limit often means that the pattern is very
	inefficient or too complex. This may already happen with the pattern
	\(.\)* on a very long line. .* works much better.
	Might also happen on redraw, when syntax rules try to match a complex
	text structure.
	Vim may run out of memory before hitting the 'maxmempattern' limit, in
	which case you get an "Out of memory" error instead.

'maxsearchcount' 'msc'

'maxsearchcount' 'msc'	number	(default 999)
		global
	Maximum number of matches shown for the search count status shm-S
	When the number of matches exceeds this value, Vim shows ">" instead
	of the exact count to keep searching fast.
	Note: larger values may impact performance.
	The value must be between 1 and 9999. See also the searchcount()
	function.

'menuitems' 'mis'

'menuitems' 'mis'	number	(default 25)
		global
	Maximum number of items to use in a menu. Used for menus that are
	generated from a list of items, e.g., the Buffers menu. Changing this
	option has no direct effect, the menu must be refreshed first.

'messagesopt' 'mopt'

'messagesopt' 'mopt'	string	(default "hit-enter,history:500,progress:c")
		global
	Option settings for outputting messages. It can consist of the
	following items. Items must be separated by a comma.

	history:{n}	Determines how many entries are remembered in the
			:messages history. The maximum value is 10000.
			Setting it to zero clears the message history.
			This item must always be present.

hit-enter	Use a hit-enter prompt when the message is longer than 'cmdheight' size.

maxheight:{n}	ui2 only. Maximum height of the expanded cmdline for message display, as a percentage of 'lines'. A message longer than this is "collapsed", with a [+x] "spill" indicator. (default: 50)

pager:{key}	ui2 only. Key that enters the message pager after an interactive : command showed a collapsed message. Use key-notation, e.g. <CR>. A literal comma must be given as "<Char-44>". Empty: no such key.

progress:{s}	Determines where to show progress messages. Valid values are:

- empty: Progress messages not shown in cmdline.
- "c": Progress messages are shown in cmdline.

timeout:{n}	ui2 only. Time in milliseconds that a message is visible in the message window.

wait:{n}	**Deprecated** with ui2. Instead of a hit-enter prompt, simply wait for {n} milliseconds so the user has a chance to read the message. Maximum {n} is 10000. Use 0 to disable the wait (user won't see any "hit-enter" messages). Ignored when "hit-enter" is present, but required when "hit-enter" is not present.

'mkspellmem' 'msm'

'mkspellmem' 'msm'	string (default "460000,2000,500")
global
Disallowed in modeline. no-modeline-option

Parameters for :mkspell. This tunes when to start compressing the word tree. Compression can be slow when there are many words, but it's needed to avoid running out of memory. The amount of memory used per word depends very much on how similar the words are, that's why this tuning is complicated.

There are three numbers, separated by commas:

```
{start},{inc},{added}
```

For most languages the uncompressed word tree fits in memory. {start} gives the amount of memory in Kbyte that can be used before any compression is done. It should be a bit smaller than the amount of memory that is available to Vim.

When going over the {start} limit the {inc} number specifies the amount of memory in Kbyte that can be allocated before another compression is done. A low number means compression is done after less words are added, which is slow. A high number means more memory will be allocated.

After doing compression, {added} times 1024 words can be added before the {inc} limit is ignored and compression is done when any extra amount of memory is needed. A low number means there is a smaller chance of hitting the {inc} limit, less memory is used but it's slower.

The languages for which these numbers are important are Italian and Hungarian. The default works for when you have about 512 Mbyte. If you have 1 Gbyte you could use:

```
set mkspellmem=900000,3000,800
```

If you have less than 512 Mbyte :mkspell may fail for some languages, no matter what you set 'mkspellmem' to.
Environment variables are expanded :set_env.

'modeline' 'ml' 'nomodeline' 'noml'

'modeline' 'ml'	boolean (default on (off for root))
local to buffer

If 'modeline' is on 'modelines' gives the number of lines that is checked for set commands. If 'modeline' is off or 'modelines' is zero no lines are checked. See modeline.

'modelineexpr' 'mle' 'nomodelineexpr' 'nomle'

'modelineexpr' 'mle'	boolean (default off)
global
Disallowed in modeline. no-modeline-option

When on allow some options that are an expression to be set in the modeline. Check the option for whether it is affected by 'modelineexpr'. Also see modeline.

'modelines' 'mls'

'modelines' 'mls'	number (default 5)
global

If 'modeline' is on 'modelines' gives the number of lines that is checked for set commands. If 'modeline' is off or 'modelines' is zero no lines are checked. See modeline.

'modifiable' 'ma' 'nomodifiable' 'noma' E21

'modifiable' 'ma'	boolean (default on)

local to buffer
When off the buffer contents cannot be changed. The 'fileformat' and 'fileencoding' options also can't be changed.
Can be reset on startup with the -M command line argument.

'modified' 'mod' 'nomodified' 'nomod'
'modified' 'mod' boolean (default off)
local to buffer local-noglobal
When on, the buffer is considered to be modified. This option is set when:
1. A change was made to the text since it was last written. Using the undo command to go back to the original text will reset the option. But undoing changes that were made before writing the buffer will set the option again, since the text is different from when it was written.
2. 'fileformat' or 'fileencoding' is different from its original value. The original value is set when the buffer is read or written. A ":set nomodified" command also resets the original values to the current values and the 'modified' option will be reset.
Similarly for 'eol' and 'bomb'.
This option is not set when a change is made to the buffer as the result of a BufNewFile, BufRead/BufReadPost, BufWritePost, FileAppendPost or VimLeave autocommand event. See gzip-example for an explanation.

When 'buftype' is "prompt", 'modified' is not implicitly set when the buffer is changed, but a user or plugin may explicitly set it.

When 'buftype' is "nowrite" or "nofile", this option may be set, but will be ignored.

Note that the text may actually be the same, e.g. 'modified' is set when using "rA" on an "A".

'more' 'nomore'
'more' boolean (default on)
global
When on, listings pause when the whole screen is filled. You will get the more-prompt. When this option is off there are no pauses, the listing continues until finished.

'mouse'

'mouse' string (default "nvi")
global
Enables mouse support. For example, to enable the mouse in Normal mode and Visual mode:

```
set mouse=nv
```

To temporarily disable mouse support, hold the shift key while using the mouse.

Mouse support can be enabled for different modes:
n Normal mode
v Visual mode
i Insert mode
c Command-line mode
h all previous modes when editing a help file
a all previous modes
r for hit-enter and more-prompt prompt

Left-click anywhere in a text buffer to place the cursor there. This works with operators too, e.g. type d then left-click to delete text from the current cursor position to the position where you clicked.

Drag the status-line or vertical separator of a window to resize it.

If enabled for "v" (Visual mode) then double-click selects word-wise, triple-click makes it line-wise, and quadruple-click makes it rectangular block-wise.

For scrolling with a mouse wheel see scroll-mouse-wheel.

Note: When enabling the mouse in a terminal, copy/paste will use the "* register if possible. See also 'clipboard'.

Related options:
'mousefocus' window focus follows mouse pointer
'mousemodel' what mouse button does which action
'mousehide' hide mouse pointer while typing text
'selectmode' whether to start Select mode or Visual mode

'mousefocus' 'mousef' 'nomousefocus' 'nomousef'
'mousefocus' 'mousef' boolean (default off)
global
The window that the mouse pointer is on is automatically activated. When changing the window layout or window focus in another way, the mouse pointer is moved to the window with keyboard focus. Off is the default because it makes using the pull down menus a little goofy, as a pointer transit may activate a window unintentionally.

'mousehide' 'mh' 'nomousehide' 'nomh'

'mousehide' 'mh' boolean (default on)
global
only in the GUI

When on, the mouse pointer is hidden when characters are typed.
The mouse pointer is restored when the mouse is moved.

'mousemodel' 'mousem'

'mousemodel' 'mousem' string (default "popup_setpos")
global

Sets the model to use for the mouse. The name mostly specifies what the right mouse button is used for:

- extend — Right mouse button extends a selection. This works like in an xterm.
- popup — Right mouse button pops up a menu. The shifted left mouse button extends a selection. This works like with Microsoft Windows.
- popup_setpos — Like "popup", but the cursor will be moved to the position where the mouse was clicked, and thus the selected operation will act upon the clicked object. If clicking inside a selection, that selection will be acted upon, i.e. no cursor move. This implies of course, that right clicking outside a selection will end Visual mode.

For a detailed description of 'mousemodel' behaviour see mouse-mode-table. Overview of what button does what for each model:

mouse	extend	popup(_setpos)
left click	place cursor	place cursor
left drag	start selection	start selection
shift-left	search word	extend selection
right click	extend selection	popup menu (place cursor)
right drag	extend selection	-
middle click	paste	paste

In the "popup" model the right mouse button produces a pop-up menu. Nvim creates a default popup-menu but you can redefine it.

Note that you can further refine the meaning of buttons with mappings. See mouse-overview. But mappings are NOT used for modeless selection.

Example:

```
map <S-LeftMouse>     <RightMouse>
map <S-LeftDrag>      <RightDrag>
map <S-LeftRelease>   <RightRelease>
map <2-S-LeftMouse>   <2-RightMouse>
map <2-S-LeftDrag>    <2-RightDrag>
map <2-S-LeftRelease> <2-RightRelease>
map <3-S-LeftMouse>   <3-RightMouse>
map <3-S-LeftDrag>    <3-RightDrag>
map <3-S-LeftRelease> <3-RightRelease>
map <4-S-LeftMouse>   <4-RightMouse>
map <4-S-LeftDrag>    <4-RightDrag>
map <4-S-LeftRelease> <4-RightRelease>
```

Mouse commands requiring the CTRL modifier can be simulated by typing the "g" key before using the mouse:

```
"g<LeftMouse>"  is "<C-LeftMouse>   (jump to tag under mouse click)
"g<RightMouse>" is "<C-RightMouse>  ("CTRL-T")
```

'mousemoveevent' 'mousemev' 'nomousemoveevent' 'nomousemev' mouse-hover

'mousemoveevent' 'mousemev' boolean (default off)
global

When on, mouse move events are delivered to the input queue and are available for mapping <MouseMove>. The default, off, avoids the mouse movement overhead except when needed.
Warning: Setting this option can make pending mappings to be aborted when the mouse is moved.

'mousescroll' E5080

'mousescroll' string (default "ver:3,hor:6")
global

This option controls the number of lines / columns to scroll by when scrolling with a mouse wheel (scroll-mouse-wheel). The option is a comma-separated list. Each part consists of a direction and a count as follows:

```
direction:count,direction:count
```

Direction is one of either "hor" or "ver". "hor" controls horizontal scrolling and "ver" controls vertical scrolling. Count sets the amount to scroll by for the given direction, it should be a non negative integer. Each direction should be set at most once. If a direction is omitted, a default value is used (6 for horizontal scrolling and 3 for vertical scrolling). You can disable mouse scrolling by using a count of 0.

Example:

```
set mousescroll=ver:5,hor:2
```

Will make Nvim scroll 5 lines at a time when scrolling vertically, and scroll 2 columns at a time when scrolling horizontally.

'mousetime' 'mouset'

'mousetime' 'mouset'	number (default 500)
global

Defines the maximum time in msec between two mouse clicks for the second click to be recognized as a multi click.

'nrformats' 'nf'

'nrformats' 'nf'	string (default "bin,hex")
local to buffer

This defines what bases Vim will consider for numbers when using the CTRL-A and CTRL-X commands for adding to and subtracting from a number respectively; see CTRL-A for more info on these commands.

alpha	If included, single alphabetical characters will be incremented or decremented. This is useful for a list with a letter index a), b), etc.

octal-nrformats

octal	If included, numbers that start with a zero will be considered to be octal. Example: Using CTRL-A on "007" results in "010".

hex	If included, numbers starting with "0x" or "0X" will be considered to be hexadecimal. Example: Using CTRL-X on "0x100" results in "0x0ff".

bin	If included, numbers starting with "0b" or "0B" will be considered to be binary. Example: Using CTRL-X on "0b1000" subtracts one, resulting in "0b0111".

unsigned	If included, numbers are recognized as unsigned. Thus a leading dash or negative sign won't be considered as part of the number. Examples:
Using CTRL-X on "2020" in "9-2020" results in "9-2019" (without "unsigned" it would become "9-2021").
Using CTRL-A on "2020" in "9-2020" results in "9-2021" (without "unsigned" it would become "9-2019").
Using CTRL-X on "0" or CTRL-A on "18446744073709551615" (2^64 - 1) has no effect, overflow is prevented.

blank	If included, treat numbers as signed or unsigned based on preceding whitespace. If a number with a leading dash has its dash immediately preceded by a non-whitespace character (i.e., not a tab or a " "), the negative sign won't be considered as part of the number. For example:
Using CTRL-A on "14" in "Carbon-14" results in "Carbon-15" (without "blank" it would become "Carbon-13").
Using CTRL-X on "8" in "Carbon -8" results in "Carbon -9" (because -8 is preceded by whitespace. If "unsigned" was set, it would result in "Carbon -7").
If this format is included, overflow is prevented as if "unsigned" were set. If both this format and "unsigned" are included, "unsigned" will take precedence.

Numbers which simply begin with a digit in the range 1-9 are always considered decimal. This also happens for numbers that are not recognized as octal or hex.

'number' 'nu' 'nonumber' 'nonu'

'number' 'nu'	boolean (default off)
local to window

Print the line number in front of each line. When the 'n' option is excluded from 'cpoptions' a wrapped line will not use the column of line numbers.
Use the 'numberwidth' option to adjust the room for the line number.
When a long, wrapped line doesn't start with the first character, '-' characters are put before the number.
For highlighting see hl-LineNr, hl-CursorLineNr, and the :sign-define "numhl" argument.

number_relativenumber

The 'relativenumber' option changes the displayed number to be relative to the cursor. Together with 'number' there are these four combinations (cursor in line 3):

'nonu'	'nu'	'nonu'	'nu'
'nornu'	'nornu'	'rnu'	'rnu'

```
|apple          |  1 apple      |  2 apple      |  2 apple
|pear           |  2 pear       |  1 pear       |  1 pear
|nobody         |  3 nobody     |  0 nobody     |3   nobody
|there          |  4 there      |  1 there      |  1 there
```

'numberwidth' 'nuw'

'numberwidth' 'nuw'	number (default 4)
local to window

Minimal number of columns to use for the line number. Only relevant when the 'number' or 'relativenumber' option is set or printing lines with a line number. Since one space is always between the number and the text, there is one less character for the number itself.
The value is the minimum width. A bigger width is used when needed to fit the highest line number in the buffer respectively the number of rows in the window, depending on whether 'number' or 'relativenumber' is set. Thus with the Vim default of 4 there is room for a line number up to 999. When the buffer has 1000 lines five columns will be used. The minimum value is 1, the maximum value is 20.

'omnifunc' 'ofu'

'omnifunc' 'ofu'	string	(default "")
			local to buffer
			Disallowed in modeline. no-modeline-option
	This option specifies a function to be used for Insert mode omni
	completion with CTRL-X CTRL-O. i_CTRL-X_CTRL-O
	See complete-functions for an explanation of how the function is
	invoked and what it should return. The value can be the name of a
	function, a lambda or a Funcref. See option-value-function for
	more information.
	This option is usually set by a filetype plugin:
	:filetype-plugin-on

'operatorfunc' 'opfunc'

'operatorfunc' 'opfunc' string	(default "")
			global
			Disallowed in modeline. no-modeline-option
	This option specifies a function to be called by the g@ operator.
	See :map-operator for more info and an example. The value can be
	the name of a function, a lambda or a Funcref. See
	option-value-function for more information.

'packlockfile' 'plf'

'packlockfile' 'plf'	string	(default "$XDG_CONFIG_HOME/nvim/nvim-pack-lock.json")
			global
			Disallowed in modeline. no-modeline-option
	Path of vim.pack-lockfile. Must be set before the first usage of any
	vim.pack function. Environment variables are expanded :set_env.

'packpath' 'pp'

'packpath' 'pp'		string	(default see 'runtimepath')
			global
			Disallowed in modeline. no-modeline-option
	Directories used to find packages.
	See packages and packages-runtimepath.
	Environment variables are expanded :set_env.

'paragraphs' 'para'

'paragraphs' 'para'	string	(default "IPLPPPQPP TPHPLIPpLpItpplpipbp")
			global
	Specifies the nroff macros that separate paragraphs. These are pairs
	of two letters (see object-motions).

'patchexpr' 'pex'

'patchexpr' 'pex'	string	(default "")
			global
			Disallowed in modeline. no-modeline-option
	Expression which is evaluated to apply a patch to a file and generate
	the resulting new version of the file. See diff-patchexpr.

'patchmode' 'pm' E205 E206

'patchmode' 'pm'	string	(default "")
			global
	When non-empty the oldest version of a file is kept. This can be used
	to keep the original version of a file if you are changing files in a
	source distribution. Only the first time that a file is written a
	copy of the original file will be kept. The name of the copy is the
	name of the original file with the string in the 'patchmode' option
	appended. This option should start with a dot. Use a string like
	".orig" or ".org". 'backupdir' must not be empty for this to work
	(Detail: The backup file is renamed to the patchmode file after the
	new file has been successfully written, that's why it must be possible
	to write a backup file). If there was no file to be backed up, an
	empty file is created.
	When the 'backupskip' pattern matches, a patchmode file is not made.
	Using 'patchmode' for compressed files appends the extension at the
	end (e.g., "file.gz.orig"), thus the resulting name isn't always
	recognized as a compressed file.
	Only normal file name characters can be used, /*?[|<> are illegal.

'path' 'pa' E343 E345 E347 E854

'path' 'pa'		string	(default ".,,")
			global or local to buffer global-local
	This is a list of directories which will be searched when using the
	gf, [f,]f, ^Wf, :find, :sfind, :tabfind and other commands,
	provided that the file being searched for has a relative path (not
	starting with "/", "./" or "../"). The directories in the 'path'
	option may be relative or absolute.

- Use commas to separate directory names:

```
set path=.,/usr/local/include,/usr/include
```

- Spaces can also be used to separate directory names. To have a

  space in a directory name, precede it with an extra backslash, and
  escape the space:

```
set path=.,/dir/with\\\ space
```

- To include a comma in a directory name precede it with an extra

backslash:

```
set path=.,/dir/with\\,comma
```

- To search relative to the directory of the current file, use:

```
set path=.
```

- To search in the current directory use an empty string between two

commas:

```
set path=,,
```

- A directory name may end in a ':' or '/'.

- Environment variables are expanded :set_env.

- When using netrw URLs can be used. For example, adding

"https://www.vim.org" will make ":find index.html" work.
- Search upwards and downwards in a directory tree using "*", "**" and

";". See file-searching for info and syntax.
- Careful with '\' characters, type two to get one in the option:

```
set path=.,c:\\include
```

Or just use '/' instead:

```
set path=.,c:/include
```

Don't forget "." or files won't even be found in the same directory as
the file!
The maximum length is limited. How much depends on the system, mostly
it is something like 256 or 1024 characters.
You can check if all the include files are found, using the value of
'path', see :checkpath.
The use of :set+= and :set-= is preferred when adding or removing
directories from the list. This avoids problems when a future version
uses another default. To remove the current directory use:

```
set path-=
```

To add the current directory use:

```
set path+=
```

To use an environment variable, you probably need to replace the
separator. Here is an example to append $INCL, in which directory
names are separated with a semicolon:

```
let &path = &path .. "," .. substitute($INCL, ';', ',', 'g')
```

Replace the ';' with a ':' or whatever separator is used. Note that
this doesn't work when $INCL contains a comma or white space.

'preserveindent' 'pi' 'nopreserveindent' 'nopi'
'preserveindent' 'pi'   boolean (default off)
local to buffer
When changing the indent of the current line, preserve as much of the
indent structure as possible. Normally the indent is replaced by a
series of tabs followed by spaces as required (unless 'expandtab' is
enabled, in which case only spaces are used). Enabling this option
means the indent will preserve as many existing characters as possible
for indenting, and only add additional tabs or spaces as required.
'expandtab' does not apply to the preserved white space, a Tab remains
a Tab.
NOTE: When using ">>" multiple times the resulting indent is a mix of
tabs and spaces. You might not like this.
Also see 'copyindent'.
Use :retab to clean up white space.

'previewheight' 'pvh'
'previewheight' 'pvh'   number (default 12)
global
Default height for a preview window. Used for :ptag and associated
commands. Used for CTRL-W_} when no count is given.

'previewpopup' 'pvp'
'previewpopup' 'pvp'   string (default "")
global

When not empty a floating window is used for commands that would open a preview window. See preview-popup.
The option is a comma-separated list of these items:
   height Height of the window. When omitted, it is derived from the content.
   width   Width of the window. When omitted, it is derived from the content.
   border One of the 'winborder' styles. When omitted, 'winborder' is used. A custom (comma separated) border cannot be given here, use 'winborder'.
The window background uses hl-NormalFloat and the border uses hl-FloatBorder.
Not used for the insert completion info, add "popup" to 'completeopt' for that.

Example:

```
set previewpopup=height:10,width:60,border:rounded
```

'previewwindow' 'pvw' 'nopreviewwindow' 'nopvw' E590
'previewwindow' 'pvw' boolean (default off)
local to window local-noglobal
Identifies the preview window. Only one window can have this option set. It's normally not set directly, but by using one of the commands :ptag, :pedit, etc.

'pumblend' 'pb'
'pumblend' 'pb' number (default 0)
global
Enables pseudo-transparency for the popup-menu. Valid values are in the range of 0 for fully opaque popupmenu (disabled) to 100 for fully transparent background. Values between 0-30 are typically most useful.

It is possible to override the level for individual highlights within the popupmenu using highlight-blend. For instance, to enable transparency but force the current selected element to be fully opaque:

```
set pumblend=15
hi PmenuSel blend=0
```

UI-dependent. Works best with RGB colors. 'termguicolors'

'pumborder'

'pumborder' string (default "")
global
Defines the default border style of popupmenu windows. See 'winborder' for valid values. hl-PmenuBorder is used for highlighting the border, and when style is "shadow" the hl-PmenuShadow and hl-PmenuShadowThrough groups are used.

This option also applies to mouse popup menus when 'mousemodel' is set to "popup" or "popup_setpos", which will display borders using the same style.

'pumheight' 'ph'
'pumheight' 'ph' number (default 0)
global
Maximum number of items to show in the popup menu (ins-completion-menu). Zero means "use available screen space".

'pummaxwidth' 'pmw'
'pummaxwidth' 'pmw' number (default 0)
global
Maximum width for the popup menu (ins-completion-menu). When zero, there is no maximum width limit, otherwise the popup menu will never be wider than this value. Truncated text will be indicated by "trunc" value of 'fillchars' option.

This option takes precedence over 'pumwidth'.

'pumwidth' 'pw'
'pumwidth' 'pw' number (default 15)
global
Minimum width for the popup menu (ins-completion-menu). If the cursor column + 'pumwidth' exceeds screen width, the popup menu is nudged to fit on the screen.

'pyxversion' 'pyx'
'pyxversion' 'pyx' number (default 3)
global
Disallowed in modeline. no-modeline-option
Specifies the python version used for pyx* functions and commands python_x. As only Python 3 is supported, this always has the value 3. Setting any other value is an error.

'quickfixtextfunc' 'qftf'
'quickfixtextfunc' 'qftf' string (default "")
global
Disallowed in modeline. no-modeline-option
This option specifies a function to be used to get the text to display in the quickfix and location list windows. This can be used to customize the information displayed in the quickfix or location window for each entry in the corresponding quickfix or location list. See quickfix-window-function for an explanation of how to write the function and an example. The value can be the name of a function, a

lambda or a Funcref. See option-value-function for more information.

It is not allowed to change text or jump to another window while evaluating 'gftf' textlock.

'quoteescape' 'qe'

'quoteescape' 'qe' string (default "\")
local to buffer

The characters that are used to escape quotes in a string. Used for objects like a', a" and a` a'.
When one of the characters in this option is found inside a string, the following character will be skipped. The default value makes the text "foo\"bar\\" considered to be one string.

'readonly' 'ro' 'noreadonly' 'noro'

'readonly' 'ro' boolean (default off)
local to buffer local-noglobal

If on, writes fail unless you use a '!'. Protects you from accidentally overwriting a file. Default on when Vim is started in read-only mode ("vim -R") or when the executable is called "view".
When using ":w!" the 'readonly' option is reset for the current buffer, unless the 'Z' flag is in 'cpoptions'.
When using the ":view" command the 'readonly' option is set for the newly edited buffer.
See 'modifiable' for disallowing changes to the buffer.

'redrawdebug' 'rdb'

'redrawdebug' 'rdb' string (default "")
global

Flags to change the way redrawing works, for debugging purposes.
Most useful with 'writedelay' set to some reasonable value.
Supports the following flags:

compositor	Indicate each redraw event handled by the compositor by briefly flashing the redrawn regions in colors indicating the redraw type. These are the highlight groups used (and their default colors):		
	RedrawDebugNormal	gui=reverse	normal redraw passed through
	RedrawDebugClear	guibg=Yellow	clear event passed through
	RedrawDebugComposed	guibg=Green	redraw event modified by the compositor (due to overlapping grids, etc)
	RedrawDebugRecompose	guibg=Red	redraw generated by the compositor itself, due to a grid being moved or deleted.
line	introduce a delay after each line drawn on the screen. When using the TUI or another single-grid UI, "compositor" gives more information and should be preferred (every line is processed as a separate event by the compositor)		
flush	introduce a delay after each "flush" event.		
nothrottle	Turn off throttling of the message grid. This is an optimization that joins many small scrolls to one larger scroll when drawing the message area (with 'display' msgsep flag active).		
invalid	Enable stricter checking (abort) of inconsistencies of the internal screen state. This is mostly useful when running nvim inside a debugger (and the test suite).		
nodelta	Send all internally redrawn cells to the UI, even if they are unchanged from the already displayed state.		

'redrawtime' 'rdt'

'redrawtime' 'rdt' number (default 2000)
global

Time in milliseconds for redrawing the display. Applies to 'hlsearch', 'inccommand', :match highlighting, syntax highlighting, and async LanguageTree:parse().
When redrawing takes more than this many milliseconds no further matches will be highlighted.
For syntax highlighting the time applies per window. When over the limit syntax highlighting is disabled until CTRL-L is used.
This is used to avoid that Vim hangs when using a very complicated pattern.

'regexpengine' 're'

'regexpengine' 're' number (default 0)
global

This selects the default regexp engine. two-engines
The possible values are:

0	automatic selection
1	old engine
2	NFA engine

Note that when using the NFA engine and the pattern contains something that is not supported the pattern will not match. This is only useful for debugging the regexp engine.
Using automatic selection enables Vim to switch the engine, if the default engine becomes too costly. E.g., when the NFA engine uses too many states. This should prevent Vim from hanging on a combination of a complex pattern with long text.

'relativenumber' 'rnu' 'norelativenumber' 'nornu'

'relativenumber' 'rnu' boolean (default off)
local to window

Show the line number relative to the line with the cursor in front of

each line. Relative line numbers help you use the count you can
precede some vertical motion commands (e.g. j k + -) with, without
having to calculate it yourself. Especially useful in combination
with other commands (e.g. y d c < > gq gw =).
When the 'n' option is excluded from 'cpoptions' a wrapped line will
not use the column of line numbers.
The 'numberwidth' option can be used to set the room used for the line
number.
When a long, wrapped line doesn't start with the first character, '-'
characters are put before the number.
See hl-LineNr and hl-CursorLineNr for the highlighting used for
the number.

The number in front of the cursor line also depends on the value of
'number', see number_relativenumber for all combinations of the two
options.

'report'

'report'		number (default 2)
		global
	Threshold for reporting number of lines changed. When the number of
	changed lines is more than 'report' a message will be given for most
	":" commands. If you want it always, set 'report' to 0.
	For the ":substitute" command the number of substitutions is used
	instead of the number of lines.

			'revins' 'ri' 'norevins' 'nori'
'revins' 'ri'		boolean (default off)
		global
	Inserting characters in Insert mode will work backwards. See "typing
	backwards" ins-reverse. This option can be toggled with the CTRL-_
	command in Insert mode, when 'allowrevins' is set.

			'rightleft' 'rl' 'norightleft' 'norl'
'rightleft' 'rl'	boolean (default off)
		local to window
	When on, display orientation becomes right-to-left, i.e., characters
	that are stored in the file appear from the right to the left.
	Using this option, it is possible to edit files for languages that
	are written from the right to the left such as Hebrew and Arabic.
	This option is per window, so it is possible to edit mixed files
	simultaneously, or to view the same file in both ways (this is
	useful whenever you have a mixed text file with both right-to-left
	and left-to-right strings so that both sets are displayed properly
	in different windows). Also see rileft.txt.

					'rightleftcmd' 'rlc'
'rightleftcmd' 'rlc'	string (default "search")
		local to window
	Each word in this option enables the command line editing to work in
	right-to-left mode for a group of commands:

		search		"/" and "?" commands

	This is useful for languages such as Hebrew, Arabic and Farsi.
	The 'rightleft' option must be set for 'rightleftcmd' to take effect.

				'ruler' 'ru' 'noruler' 'noru'
'ruler' 'ru'		boolean (default on)
		global
	When on, show some buffer information in the ruler. Each window has
	its own ruler. The ruler of the active window is shown in the last
	line of the screen, unless 'cmdheight' is zero, or the window has a
	status line, which (by default) contains the window's ruler.

	'rulerformat' determines the contents of the ruler.
	By default, the line and column number of the current cursor position
	are shown, separated by a comma.
	If the number of characters displayed is different from the number of
	bytes in the text (e.g., for a TAB or a multibyte character), both
	the text column (byte number) and the screen column are shown,
	separated with a dash.
	For an empty line "0-1" is shown.
	For an empty buffer the line number will also be zero: "0,0-1".
	Finally, the relative position of the displayed text in the file is
	shown on the far right:
		Top	first line is visible
		Bot	last line is visible
		All	first and last line are visible
		45%	relative position in the file

	If you don't want to see the ruler all the time but want to know where
	you are, use "g CTRL-G" g_CTRL-G.
	When 'ruler' is on, the output of CTRL-G doen't contain the current
	line number of the cursor position.

					'rulerformat' 'ruf'
'rulerformat' 'ruf'	string (default "%18(%l,%c%V%= %P%)%<")
		global
	This option determines the content of the ruler string, as displayed
	for the 'ruler' option.
	The format of this option is like that of 'statusline'.
	Setting to empty (:set rulerformat=) sets the value to the default.
	This option cannot be set in a modeline when 'modelineexpr' is off.

	When 'ruler' is on, the default 'statusline' includes 'rulerformat'.

The default ruler width is 18 characters, which is configured with what looks like an item group. But there are some differences in this particular case. Most notably, the width is fixed and not a minimum, and the ruler is left-aligned, whereas the alignment of item groups is configurable and right-aligned by default.

When ui2 is enabled, the ruler no longer has a fixed width and the item group syntax has no special meaning for 'rulerformat'.

'runtimepath' 'rtp' vimfiles

'runtimepath' 'rtp' string (default "$XDG_CONFIG_HOME/nvim,
$XDG_CONFIG_DIRS[1]/nvim,
$XDG_CONFIG_DIRS[2]/nvim,
…
$XDG_DATA_HOME/nvim[-data]/site,
$XDG_DATA_DIRS[1]/nvim/site,
$XDG_DATA_DIRS[2]/nvim/site,
…
$VIMRUNTIME,
…
$XDG_DATA_DIRS[2]/nvim/site/after,
$XDG_DATA_DIRS[1]/nvim/site/after,
$XDG_DATA_HOME/nvim[-data]/site/after,
…
$XDG_CONFIG_DIRS[2]/nvim/after,
$XDG_CONFIG_DIRS[1]/nvim/after,
$XDG_CONFIG_HOME/nvim/after")
global
Disallowed in modeline. no-modeline-option

List of directories to be searched for these runtime files:

filetype.lua	filetypes new-filetype
autoload/	automatically loaded scripts autoload-functions
colors/	color scheme files :colorscheme
compiler/	compiler files :compiler
doc/	documentation write-local-help
ftplugin/	filetype plugins write-filetype-plugin
indent/	indent scripts indent-expression
keymap/	key mapping files mbyte-keymap
lang/	menu translations :menutrans
lsp/	LSP client configurations lsp-config
lua/	Lua plugins
menu.vim	GUI menus menu.vim
pack/	packages :packadd
parser/	treesitter syntax parsers
plugin/	plugin scripts write-plugin
queries/	treesitter queries
rplugin/	remote-plugin scripts
spell/	spell checking files spell
syntax/	syntax files mysyntaxfile
tutor/	tutorial files :Tutor

And any other file searched for with the :runtime command.

Defaults are setup to search these locations:

1. Your home directory, for personal preferences.

   Given by stdpath("config"). $XDG_CONFIG_HOME
2. Directories which must contain configuration files according to

   xdg ($XDG_CONFIG_DIRS, defaults to /etc/xdg). This also contains preferences from system administrator.
3. Data home directory, for plugins installed by user.

   Given by stdpath("data")/site. $XDG_DATA_HOME
4. nvim/site subdirectories for each directory in $XDG_DATA_DIRS.

   This is for plugins which were installed by system administrator, but are not part of the Nvim distribution. XDG_DATA_DIRS defaults to /usr/local/share/:/usr/share/, so system administrators are expected to install site plugins to /usr/share/nvim/site.
5. $VIMRUNTIME, for files distributed with Nvim.

after-directory

   6, 7, 8, 9. In after/ subdirectories of 1, 2, 3 and 4, with reverse ordering. This is for preferences to overrule or add to the distributed defaults or system-wide settings (rarely needed).

packages-runtimepath

"start" packages will also be searched (runtime-search-path) for runtime files after these, though such packages are not explicitly reported in &runtimepath. But "opt" packages are explicitly added to &runtimepath by :packadd.

Note that, unlike 'path', no wildcards like "**" are allowed. Normal wildcards are allowed, but can significantly slow down searching for runtime files. For speed, use as few items as possible and avoid

wildcards.
See :runtime.
Example:

```
set runtimepath=~/vimruntime,/mygroup/vim,$VIMRUNTIME
```

This will use the directory "~/vimruntime" first (containing your
personal Nvim runtime files), then "/mygroup/vim", and finally
"$VIMRUNTIME" (the default runtime files).
You can put a directory before $VIMRUNTIME to find files which replace
distributed runtime files. You can put a directory after $VIMRUNTIME
to find files which add to distributed runtime files.

With --clean the home directory entries are not included.
Environment variables are expanded :set_env.

'scroll' 'scr'
'scroll' 'scr' number (default half the window height)
local to window local-noglobal

Number of lines to scroll with CTRL-U and CTRL-D commands. Will be
set to half the number of lines in the window when the window size
changes. This may happen when enabling the status-line or
'tabline' option after setting the 'scroll' option.
If you give a count to the CTRL-U or CTRL-D command it will
be used as the new value for 'scroll'. Reset to half the window
height with ":set scroll=0".

'scrollback' 'scbk'
'scrollback' 'scbk' number (default 10000)
local to buffer

Maximum number of lines kept beyond the visible screen. Lines at the
top are deleted if new lines exceed this limit.
Minimum is 1, maximum is 1000000.
Only in terminal and prompt-buffer buffers.

Note: Lines that are not visible and kept in terminal scrollback are not
reflown when the terminal buffer is resized horizontally.

'scrollbind' 'scb' 'noscrollbind' 'noscb'
'scrollbind' 'scb' boolean (default off)
local to window

Enables synchronized scrolling (in all windows with this option set).
Useful for comparing two versions of a file, see 'diff'.
Behavior is controlled by 'scrollopt'. See scroll-binding.

This option is (usually) reset when splitting a window to edit another
file: ":split | edit file" results in two windows with scroll-binding,
but ":split file" does not.

Note: Consider calling :syncbind on WinResized, WinEnter events
(scoped to relevant buffers).

'scrolljump' 'sj'
'scrolljump' 'sj' number (default 1)
global

Minimal number of lines to scroll when the cursor gets off the
screen (e.g., with "j"). Not used for scroll commands (e.g., CTRL-E,
CTRL-D). Useful if your terminal scrolls very slowly.
When set to a negative number from -1 to -100 this is used as the
percentage of the window height. Thus -50 scrolls half the window
height.

'scrolloff' 'so'
'scrolloff' 'so' number (default 0)
global or local to window global-local

Minimal number of screen lines to keep above and below the cursor.
This will make some context visible around where you are working. If
you set it to a very large value (999) the cursor line will always be
in the middle of the window (except at the start or end of the file,
see 'scrolloffpad', or when long lines wrap).
After using the local value, go back the global value with one of
these two:

```
setlocal scrolloff<
setlocal scrolloff=-1
```

For scrolling horizontally see 'sidescrolloff'.

'scrolloffpad' 'sop'
'scrolloffpad' 'sop' number (default 0)
global or local to window global-local

When 'scrolloff' and 'scrolloffpad' are greater than zero, allow
the cursor to remain centered when at the end of the file.
Normally, 'scrolloff' will not keep the cursor centered at the
end of the file.

A value of 0 disables this feature. Any value above 0 enables it.
For a window-local value, -1 means to use the global value.
Values below -1 are invalid.

Example:

```
:set scrolloff=99 scrolloffpad=1
```

After using the local value, go back the global value with one of
these two:

```
setlocal scrolloffpad<
setlocal scrolloffpad=-1
```

'scrollopt' 'sbo'

'scrollopt' 'sbo' string (default "ver,jump")
global
This is a comma-separated list of words that specifies how 'scrollbind' windows should behave. 'sbo' stands for ScrollBind Options.
The following words are available:
ver Bind vertical scrolling for 'scrollbind' windows
hor Bind horizontal scrolling for 'scrollbind' windows
jump Applies to the offset between two windows for vertical scrolling. This offset is the difference in the first displayed line of the bound windows. When moving around in a window, another 'scrollbind' window may reach a position before the start or after the end of the buffer. The offset is not changed though, when moving back the 'scrollbind' window will try to scroll to the desired position when possible.
When now making that window the current one, two things can be done with the relative offset:

1. When "jump" is not included, the relative offset is adjusted for the scroll position in the new current window. When going back to the other window, the new relative offset will be used.
2. When "jump" is included, the other windows are scrolled to keep the same relative offset. When going back to the other window, it still uses the same relative offset.

Also see scroll-binding.
When 'diff' mode is active there always is vertical scroll binding, even when "ver" isn't there.

'sections' 'sect'

'sections' 'sect' string (default "SHNHH HUnhsh")
global
Specifies the nroff macros that separate sections. These are pairs of two letters (See object-motions). The default makes a section start at the nroff macros ".SH", ".NH", ".H", ".HU", ".nh" and ".sh".

'selection' 'sel'

'selection' 'sel' string (default "inclusive")
global
This option defines the behavior of the selection. It is only used in Visual and Select mode.
Possible values:

value	past line	inclusive
old	no	yes
inclusive	yes	yes
exclusive	yes	no

"past line" means that the cursor is allowed to be positioned one character past the line.
"inclusive" means that the last character of the selection is included in an operation. For example, when "x" is used to delete the selection.
When "old" is used and 'virtualedit' allows the cursor to move past the end of line the line break still isn't included.
When "exclusive" is used, cursor position in visual mode will be adjusted for inclusive motions inclusive-motion-selection-exclusive.

Note:

- When "exclusive" is used and selecting from the end backwards, you cannot include the last character of a line, when starting in Normal mode and 'virtualedit' empty.
- when "exclusive" is used with a single character visual selection, Vim will behave as if the 'selection' is inclusive (in other words, you cannot visually select an empty region).

'selectmode' 'slm'

'selectmode' 'slm' string (default "")
global
This is a comma-separated list of words, which specifies when to start Select mode instead of Visual mode, when a selection is started.
Possible values:
mouse when using the mouse
key when using shifted special keys
cmd when using "v", "V" or CTRL-V
See Select-mode.

'sessionoptions' 'ssop'

'sessionoptions' 'ssop' string (default "blank,buffers,curdir,folds,help,tabpages,winsize,terminal")
global
Changes the effect of the :mksession command. It is a comma-

separated list of words. Each word enables saving and restoring something:

word	save and restore
blank	empty windows
buffers	hidden and unloaded buffers, not just those in windows
curdir	the current directory
folds	manually created folds, opened/closed folds and local fold options
globals	global variables that start with an uppercase letter and contain at least one lowercase letter. Only String and Number types are stored.
help	the help window
localoptions	options and mappings local to a window or buffer (not global values for local options)
options	all options and mappings (also global values for local options), except Lua functions option-value-function.
skiprtp	exclude 'runtimepath' and 'packpath' from the options
resize	size of the Vim window: 'lines' and 'columns'
sesdir	the directory in which the session file is located will become the current directory (useful with projects accessed over a network from different systems)
tabpages	all tabpages; without this only the current tabpage is restored, so that you can make a session for each tabpage separately
terminal	include terminal windows where the command can be restored
winpos	position of the whole Vim window
winsize	window sizes
slash	deprecated Always enabled. Uses "/" in filenames.
unix	deprecated Always enabled. Uses "\n" line endings.

Don't include both "curdir" and "sesdir". When neither is included filenames are stored as absolute paths.
If you leave out "options" many things won't work well after restoring the session.

'shada' 'sd' E526 E527 E528

'shada' 'sd' string (default "!,'100,<50,s10,h,r/tmp/,r/private/")
global
Disallowed in modeline. no-modeline-option

When non-empty, the shada file is read upon startup and written when exiting Vim (see shada-file). The string should be a comma-separated list of parameters, each consisting of a single character identifying the particular parameter, followed by a number or string which specifies the value of that parameter. If a particular character is left out, then the default value is used for that parameter. The following is a list of the identifying characters and the effect of their value.

CHAR	VALUE
shada-!	
!	When included, save and restore global variables that start with an uppercase letter, and don't contain a lowercase letter. Thus "KEEPTHIS and "K_L_M" are stored, but "KeepThis" and "_K_L_M" are not. Nested List and Dict items may not be read back correctly, you end up with an empty item.
shada-quote	
"	Maximum number of lines saved for each register. Old name of the '<' item, with the disadvantage that you need to put a backslash before the ", otherwise it will be recognized as the start of a comment!
shada-%	
%	When included, save and restore the buffer list. If Vim is started with a file name argument, the buffer list is not restored. If Vim is started without a file name argument, the buffer list is restored from the shada file. Quickfix ('buftype'), unlisted ('buflisted'), unnamed and buffers on removable media (shada-r) are not saved. When followed by a number, the number specifies the maximum number of buffers that are stored. Without a number all buffers are stored.
shada-'	
'	Maximum number of previously edited files for which the marks are remembered. This parameter must always be included when 'shada' is non-empty. If non-zero, then the jumplist and the changelist are also stored in the shada file.
shada-/	
/	Maximum number of items in the search pattern history to be saved. If non-zero, then the previous search and substitute patterns are also saved. When not included, the value of 'history' is used.
shada-:	
:	Maximum number of items in the command-line history to be saved. When not included, the value of 'history' is used.

shada-<

<	Maximum number of lines saved for each register. If zero then
	registers are not saved. When not included, all lines are
	saved. '"' is the old name for this item.
	Also see the 's' item below: limit specified in KiB.

shada-@

@	Maximum number of items in the input-line history to be
	saved. When not included, the value of 'history' is used.

shada-c

c	Dummy option, kept for compatibility reasons. Has no actual
	effect: ShaDa always uses UTF-8 and 'encoding' value is fixed
	to UTF-8 as well.

shada-f

f	Whether file marks need to be stored. If zero, file marks ('0
	to '9, 'A to 'Z) are not stored. When not present or when
	non-zero, they are all stored. '0 is used for the current
	cursor position (when exiting or when doing :wshada).

shada-h

h	Disable the effect of 'hlsearch' when loading the shada
	file. When not included, it depends on whether ":nohlsearch"
	has been used since the last search command.

shada-n

n	Name of the shada file. The name must immediately follow
	the 'n'. Must be at the end of the option! If the
	'shadafile' option is set, that file name overrides the one
	given here with 'shada'. Environment variables are
	expanded when opening the file, not when setting the option.

shada-r

r	Removable media. The argument is a string (up to the next
	','). This parameter can be given several times. Each
	specifies the start of a path for which no marks will be
	stored. This is to avoid removable media. For Windows you
	could use "ra:,rb:". You can also use it for temp files,
	e.g., for Unix: "r/tmp". Case is ignored.

shada-s

s	Maximum size of an item contents in KiB. If zero then nothing
	is saved. Unlike Vim this applies to all items, except for
	the buffer list and header. Full item size is off by three
	unsigned integers: with s10 maximum item size may be 1 byte
	(type: 7-bit integer) + 9 bytes (timestamp: up to 64-bit
	integer) + 3 bytes (item size: up to 16-bit integer because
	2^8 < 10240 < 2^16) + 10240 bytes (requested maximum item
	contents size) = 10253 bytes.

Example:

```
set shada='50,<1000,s100,:0,n~/nvim/shada
```

	'50		Marks will be remembered for the last 50 files you
			edited.
	<1000		Contents of registers (up to 1000 lines each) will be
			remembered.
	s100		Items with contents occupying more then 100 KiB are
			skipped.
	:0		Command-line history will not be saved.
	n~/nvim/shada	The name of the file to use is "~/nvim/shada".
	no /		Since '/' is not specified, the default will be used,
			that is, save all of the search history, and also the
			previous search and substitute patterns.
	no %		The buffer list will not be saved nor read back.
	no h		'hlsearch' highlighting will be restored.

When setting 'shada' from an empty value you can use :rshada to
load the contents of the file, this is not done automatically.

'shadafile' 'sdf'

'shadafile' 'sdf'	string (default "")
			global
			Disallowed in modeline. no-modeline-option
When non-empty, overrides the file name used for shada (viminfo).
When equal to "NONE" no shada file will be read or written.
This option can be set with the -i command line flag. The --clean
command line flag sets it to "NONE".
Environment variables are expanded :set_env.

'shell' 'sh' E91

'shell' 'sh'		string (default $SHELL or "sh", Win32: "cmd.exe")
			global
			Disallowed in modeline. no-modeline-option
Name of the shell to use for ! and :! commands. When changing the
value also check these options: 'shellpipe', 'shellslash'
'shellredir', 'shellquote', 'shellxquote' and 'shellcmdflag'.
It is allowed to give an argument to the command, e.g. "csh -f".

See option-backslash about including spaces and backslashes.
Environment variables are expanded :set_env.

If the name of the shell contains a space, you need to enclose it in quotes. Example with quotes:

```
set shell=\"c:\program\ files\unix\sh.exe\"\ -f
```

Note the backslash before each quote (to avoid starting a comment) and each space (to avoid ending the option value), so better use :let-& like this:

```
let &shell='"C:\Program Files\unix\sh.exe" -f'
```

Also note that the "-f" is not inside the quotes, because it is not part of the command name.

shell-unquoting

Rules regarding quotes:

1. Option is split on space and tab characters that are not inside quotes: "abc def" runs shell named "abc" with additional argument "def", '"abc def"' runs shell named "abc def" with no additional arguments (here and below: additional means “additional to 'shellcmdflag'”).
2. Quotes in option may be present in any position and any number: '"abc"', '"a"bc', 'a"b"c', 'ab"c"' and '"a"b"c"' are all equivalent to just "abc".
3. Inside quotes backslash preceding backslash means one backslash. Backslash preceding quote means one quote. Backslash preceding anything else means backslash and next character literally: '"a\\b"' is the same as "a\b", '"a\"b"' runs shell named literally 'a"b', '"a\b"' is the same as "a\b" again.
4. Outside of quotes backslash always means itself, it cannot be used to escape quote: 'a\"b"' is the same as "a\b".

Note that such processing is done after :set did its own round of unescaping, so to keep yourself sane use :let-& like shown above.

shell-powershell

To use PowerShell:

```
set noshelltemp
let &shell = 'powershell'
let &shellcmdflag = '-NoLogo -NoProfile -ExecutionPolicy RemoteSigned -Command '
let &shellcmdflag .= '[Console]::InputEncoding=[Console]::OutputEncoding=[System.Text.UTF8Encoding]::new();'
let &shellcmdflag .= '$PSDefaultParameterValues[''Out-File:Encoding'']=''utf8'';'
let &shellpipe  = '> %s 2>&1'
set shellquote= shellxquote=
```

shell-pwsh

To use pwsh, use the above settings with let &shell = 'pwsh', and add:

```
let &shellcmdflag .= '$PSStyle.OutputRendering = ''PlainText'';'
" Workaround (may not be needed in future version of pwsh):
let $__SuppressAnsiEscapeSequences = 1
```

'shellcmdflag' 'shcf'

'shellcmdflag' 'shcf' string (default "-c"; Windows, when 'shell' contains "cmd" somewhere: "/s /c")
global
Disallowed in modeline. no-modeline-option
Flag passed to the shell to execute "!" and ":!" commands; e.g., bash.exe -c ls or cmd.exe /s /c "dir". For MS-Windows, the default is set according to the value of 'shell', to reduce the need to set this option by the user.
On Unix it can have more than one flag. Each white space separated part is passed as an argument to the shell command.
See option-backslash about including spaces and backslashes.
See shell-unquoting which talks about separating this option into multiple arguments.

'shellpipe' 'sp'

'shellpipe' 'sp' string (default ">", "| tee", "|& tee" or "2>&1| tee")
global
Disallowed in modeline. no-modeline-option
String to be used to put the output of the ":make" command in the error file. See also :make_makeprg. See option-backslash about including spaces and backslashes.
The name of the temporary file can be represented by "%s" if necessary (the file name is appended automatically if no %s appears in the value of this option).

For MS-Windows the default is "2>&1| tee". The stdout and stderr are
saved in a file and echoed to the screen.
For Unix the default is "| tee". The stdout of the compiler is saved
in a file and echoed to the screen. If the 'shell' option contains
"csh" (e.g. "tcsh") after initializations, the default becomes
"|& tee". Otherwise, if it contains "sh" (e.g. "bash", "zsh"), the
default becomes "2>&1| tee". This means that stderr is also included.
Before using the 'shell' option a path is removed, thus "/bin/sh" uses
"sh".
The initialization of this option is done after reading the vimrc
and the other initializations, so that when the 'shell' option is set
there, the 'shellpipe' option changes automatically, unless it was
explicitly set before.
When 'shellpipe' is set to an empty string, no redirection of the
":make" output will be done. This is useful if you use a 'makeprg'
that writes to 'makeef' by itself. If you want no piping, but do
want to include the 'makeef', set 'shellpipe' to a single space.
Don't forget to precede the space with a backslash: ":set sp=\ ".
In the future pipes may be used for filtering and this option will
become obsolete (at least for Unix).
Note: When using a pipe like "| tee", you'll lose the exit code of the
shell command. This might be configurable by your shell, look for
the pipefail option (for bash and zsh, use ":set -o pipefail").
Only a single "%s" value is allowed.

'shellquote' 'shq'
'shellquote' 'shq'	string (default ""; Windows, when 'shell'
contains "sh" somewhere: "\"")
global
Disallowed in modeline. no-modeline-option
Quoting character(s), put around the command passed to the shell, for
the "!" and ":!" commands. The redirection is kept outside of the
quoting. See 'shellxquote' to include the redirection. It's
probably not useful to set both options.
This is an empty string by default. Only known to be useful for
third-party shells on Windows systems, such as the MKS Korn Shell
or bash, where it should be "\"". The default is adjusted according
the value of 'shell', to reduce the need to set this option by the
user.

'shellredir' 'srr'
'shellredir' 'srr'	string (default ">", ">&" or ">%s 2>&1")
global
Disallowed in modeline. no-modeline-option
String to be used to put the output of a filter command in a temporary
file. See also :!. See option-backslash about including spaces
and backslashes.
The name of the temporary file can be represented by "%s" if necessary
(the file name is appended automatically if no %s appears in the value
of this option).
The default is ">". For Unix, if the 'shell' option contains "csh"
(e.g. "tcsh") during initializations, the default becomes ">&".
Otherwise, if it contains "sh" (e.g. "bash", "zsh"), the default
becomes ">%s 2>&1". This means that stderr is also included. For
Win32, the Unix checks are done and additionally "cmd" is checked
for, which makes the default ">%s 2>&1".
Also, the same names with ".exe" appended are checked for.
The initialization of this option is done after reading the vimrc
and the other initializations, so that when the 'shell' option is set
there, the 'shellredir' option changes automatically unless it was
explicitly set before.
In the future pipes may be used for filtering and this option will
become obsolete (at least for Unix).

E1577

Only a single "%s" item is allowed in the option value.

'shellslash' 'ssl' 'noshellslash' 'nossl'
'shellslash' 'ssl'	boolean (default on; Windows: off, except when 'shell'
contains "sh" somewhere)
global
only modifiable in MS-Windows
When set, a forward slash is used when expanding file names. This is
useful when a Unix-like shell is used instead of cmd.exe. Backward
slashes can still be typed, but they are changed to forward slashes by
Vim.
Note that setting or resetting this option has no effect for some
existing file names, thus this option needs to be set before opening
any file for best results. This might change in the future.
'shellslash' only works when a backslash can be used as a path
separator. To test if this is so use:

```
if exists('+shellslash')
```

Also see 'completeslash'.

'shelltemp' 'stmp' 'noshelltemp' 'nostmp'
'shelltemp' 'stmp'	boolean (default off)
global
When on, use temp files for shell commands. When off use a pipe.
When using a pipe is not possible temp files are used anyway.
The advantage of using a pipe is that nobody can read the temp file

and the 'shell' command does not need to support redirection.
The advantage of using a temp file is that the file type and encoding
can be detected.
The FilterReadPre, FilterReadPost and FilterWritePre,
FilterWritePost autocommands event are not triggered when
'shelltemp' is off.
system() does not respect this option, it always uses pipes.

'shellxescape' 'sxe'
'shellxescape' 'sxe'	string (default "")
global
Disallowed in modeline. no-modeline-option
When 'shellxquote' is set to "(" then the characters listed in this
option will be escaped with a '^' character. This makes it possible
to execute most external commands with cmd.exe.

'shellxquote' 'sxq'
'shellxquote' 'sxq'	string (default ""; Windows, when 'shell'
contains "cmd" somewhere: "\"")
global
Disallowed in modeline. no-modeline-option
Quoting character(s), put around the command passed to the shell, for
the "!" and ":!" commands. Includes the redirection. See
'shellquote' to exclude the redirection. It's probably not useful
to set both options.
When the value is '(' then ')' is appended. When the value is '"('
then ')"' is appended.
When the value is '(' then also see 'shellxescape'.

'shiftround' 'sr' 'noshiftround' 'nosr'
'shiftround' 'sr'	boolean (default off)
global
Round indent to multiple of 'shiftwidth'. Applies to > and <
commands. CTRL-T and CTRL-D in Insert mode always round the indent to
a multiple of 'shiftwidth' (this is Vi compatible).

'shiftwidth' 'sw'
'shiftwidth' 'sw'	number (default 8)
local to buffer
Number of columns that make up one level of (auto)indentation. Used
by 'cindent', <<, >>, etc.
If set to 0, Vim uses the current 'tabstop' value. Use shiftwidth()
to obtain the effective value in scripts.

'shortmess' 'shm' E1336
'shortmess' 'shm'	string (default "ltToOCF")
global
Controls display of file messages (e.g. CTRL-G) and various other
messages.

It is a list of flags:

flag	meaning when present	
l	use "999L, 888B" instead of "999 lines, 888 bytes"	shm-l
m	use "[+]" instead of "[Modified]"	shm-m
r	use "[RO]" instead of "[readonly]"	shm-r
w	use "[w]" instead of "written" for file write message and "[a]" instead of "appended" for ':w >> file' command	shm-w
a	all of the above abbreviations	shm-a
o	overwrite message for writing a file with subsequent message for reading a file (useful for ":wn" or when 'autowrite' on)	shm-o
O	message for reading a file overwrites any previous message; also for quickfix message (e.g., ":cn")	shm-O
s	don't give "search hit BOTTOM, continuing at TOP" or "search hit TOP, continuing at BOTTOM" messages; when using the search count do not show "W" before the count message (see shm-S below)	shm-s
t	truncate file message at the start if it is too long to fit on the command-line, "<" will appear in the left most column; ignored in Ex mode	shm-t
T	truncate other messages in the middle if they are too long to fit on the command line; "..." will appear in the middle; ignored in Ex mode	shm-T
W	don't give "written" or "[w]" when writing a file	shm-W
A	don't give the "ATTENTION" message when an existing swap file is found	shm-A
I	don't give the intro message when starting Vim, see :intro	shm-I
c	don't give ins-completion-menu messages; for example, "-- XXX completion (YYY)", "match 1 of 2", "The only match", "Pattern not found", "Back at original", etc.	shm-c
C	don't give messages while scanning for ins-completion items, for instance "scanning tags"	shm-C
q	do not show "recording @a" when recording a macro	shm-q
F	don't give the file info when editing a file, like :silent was used for the command; note that this also affects messages from 'autoread' reloading	shm-F
S	do not show search count message when searching, e.g. "[1/5]". When the "S" flag is not present (e.g. search count	shm-S

is shown), the "search hit BOTTOM, continuing at TOP" and
"search hit TOP, continuing at BOTTOM" messages are only
indicated by a "W" (Mnemonic: Wrapped) letter before the
search count statistics. The maximum limit can be set with
the 'maxsearchcount' option, see also searchcount()
function.
u don't give undo and redo messages like shm-u

"1 line less; before #1 1 second ago", "Already at oldest
change" or "Already at newest change"

This gives you the opportunity to avoid that a change between buffers
requires you to hit <Enter>, but still gives as useful a message as
possible for the space available. To get the whole message that you
would have got with 'shm' empty, use ":file!"
Useful values:
shm= No abbreviation of message.
shm=a Abbreviation, but no loss of information.
shm=at Abbreviation, and truncate message when necessary.

'showbreak' 'sbr' E595
'showbreak' 'sbr' string (default "")
global or local to window global-local
String to put at the start of lines that have been wrapped. Useful
values are "> " or "+++ ":

```
let &showbreak = "> "
let &showbreak = '+++ '
```

Only printable single-cell characters are allowed, excluding <Tab> and
comma (in a future version the comma might be used to separate the
part that is shown at the end and at the start of a line).
The hl-NonText highlight group determines the highlighting.
Note that tabs after the showbreak will be displayed differently.
If you want the 'showbreak' to appear in between line numbers, add the
"n" flag to 'cpoptions'.
A window-local value overrules a global value. If the global value is
set and you want no value in the current window use NONE:

```
setlocal showbreak=NONE
```

'showcmd' 'sc' 'noshowcmd' 'nosc'
'showcmd' 'sc' boolean (default on)
global
Show (partial) command in the last line of the screen. Set this
option off if your terminal is slow.
In Visual mode the size of the selected area is shown:

- When selecting characters within a line, the number of characters.

  If the number of bytes is different it is also displayed: "2-6"
  means two characters and six bytes.
- When selecting more than one line, the number of lines.

- When selecting a block, the size in screen characters:

  {lines}x{columns}.
This information can be displayed in an alternative location using the
'showcmdloc' option, useful when 'cmdheight' is 0.

'showcmdloc' 'sloc'
'showcmdloc' 'sloc' string (default "last")
global
This option can be used to display the (partially) entered command in
another location. Possible values are:
last Last line of the screen (default).
statusline Status line of the current window.
tabline First line of the screen if 'showtabline' is enabled.
Setting this option to "statusline" or "tabline" means that these will
be redrawn whenever the command changes, which can be on every key
pressed.
The %S 'statusline' item can be used in 'statusline' or 'tabline' to
place the text. Without a custom 'statusline' or 'tabline' it will be
displayed in a convenient location.

'showfulltag' 'sft' 'noshowfulltag' 'nosft'
'showfulltag' 'sft' boolean (default off)
global
When completing a word in insert mode (see ins-completion) from the
tags file, show both the tag name and a tidied-up form of the search
pattern (if there is one) as possible matches. Thus, if you have
matched a C function, you can see a template for what arguments are
required (coding style permitting).
Note that this doesn't work well together with having "longest" in
'completeopt', because the completion from the search pattern may not
match the typed text.

'showmatch' 'sm' 'noshowmatch' 'nosm'
'showmatch' 'sm' boolean (default off)
global

When a bracket is inserted, briefly jump to the matching one. The jump is only done if the match can be seen on the screen. The time to show the match can be set with 'matchtime'.
A Beep is given if there is no match (no matter if the match can be seen or not).
When the 'm' flag is not included in 'cpoptions', typing a character will immediately move the cursor back to where it belongs.
See the "sm" field in 'guicursor' for setting the cursor shape and blinking when showing the match.
The 'matchpairs' option can be used to specify the characters to show matches for. 'rightleft' and 'revins' are used to look for opposite matches.
Also see the matchparen plugin for highlighting the match when moving around pi_paren.txt.
Note: Use of the short form is rated PG.

'showmode' 'smd' 'noshowmode' 'nosmd'
'showmode' 'smd' boolean (default on)
global
If in Insert, Replace or Visual mode put a message on the last line.
The hl-ModeMsg highlight group determines the highlighting.
The option has no effect when 'cmdheight' is zero.

'showtabline' 'stal'
'showtabline' 'stal' number (default 1)
global
Specifies when the tabpage labels will be displayed:
0: never
1: only if there are at least two tabpages
2: always

'sidescroll' 'ss'
'sidescroll' 'ss' number (default 1)
global
The minimal number of columns to scroll horizontally. Used only when the 'wrap' option is off and the cursor is moved off of the screen.
When it is zero the cursor will be put in the middle of the screen.
When using a slow terminal set it to a large number or 0. Not used for "zh" and "zl" commands.

'sidescrolloff' 'siso'
'sidescrolloff' 'siso' number (default 0)
global or local to window global-local
The minimal number of screen columns to keep to the left and to the right of the cursor if 'nowrap' is set. Setting this option to a value greater than 0 while having 'sidescroll' also at a non-zero value makes some context visible in the line you are scrolling in horizontally (except at beginning of the line). Setting this option to a large value (like 999) has the effect of keeping the cursor horizontally centered in the window, as long as one does not come too close to the beginning of the line.
After using the local value, go back the global value with one of these two:

```
setlocal sidescrolloff<
setlocal sidescrolloff=-1
```

Example: Try this together with 'sidescroll' and 'listchars' as in the following example to never allow the cursor to move onto the "extends" character:

```
set nowrap sidescroll=1 listchars=extends:>,precedes:<
set sidescrolloff=1
```

'signcolumn' 'scl'
'signcolumn' 'scl' string (default "auto")
local to window
When and how to draw the signcolumn. Valid values are:
"auto" only when there is a sign to display
"auto:[1-9]" resize to accommodate multiple signs up to the given number (maximum 9), e.g. "auto:4"
"auto:[1-8]-[2-9]" resize to accommodate multiple signs up to the given maximum number (maximum 9) while keeping at least the given minimum (maximum 8) fixed space. The minimum number should always be less than the maximum number, e.g. "auto:2-5"
"no" never
"yes" always
"yes:[1-9]" always, with fixed space for signs up to the given number (maximum 9), e.g. "yes:3"
"number" display signs in the 'number' column. If the number column is not present, then behaves like "auto".

'smartcase' 'scs' 'nosmartcase' 'noscs'
'smartcase' 'scs' boolean (default off)
global
Override the 'ignorecase' option if the search pattern contains upper case characters. Only used when the search pattern is typed and 'ignorecase' option is on. Used for the commands "/", "?", "n", "N", ":g" and ":s" and when filtering matches for the completion menu

compl-states.
Not used for "*", "#", "gd", tag search, etc. After "*" and "#" you can make 'smartcase' used by doing a "/" command, recalling the search pattern from history and hitting <Enter>.

'smartindent' 'si' 'nosmartindent' 'nosi'
'smartindent' 'si'	boolean (default off)
local to buffer

Do smart autoindenting when starting a new line. Works for C-like programs, but can also be used for other languages. 'cindent' does something like this, works better in most cases, but is more strict, see C-indenting. When 'cindent' is on or 'indentexpr' is set, setting 'si' has no effect. 'indentexpr' is a more advanced alternative.
Normally 'autoindent' should also be on when using 'smartindent'.
An indent is automatically inserted:

- After a line ending in "{".
- After a line starting with a keyword from 'cinwords'.
- Before a line starting with "}" (only with the "O" command).

When typing '}' as the first character in a new line, that line is given the same indent as the matching "{".
When typing '#' as the first character in a new line, the indent for that line is removed, the '#' is put in the first column. The indent is restored for the next line. If you don't want this, use this mapping: ":inoremap # X^H#", where ^H is entered with CTRL-V CTRL-H.
When using the ">>" command, lines starting with '#' are not shifted right.

'smarttab' 'sta' 'nosmarttab' 'nosta'
'smarttab' 'sta'	boolean (default on)
global

When enabled, the <Tab> key will indent by 'shiftwidth' if the cursor is in leading whitespace. The <BS> key has the opposite effect.
In leading whitespace, this has the same effect as setting 'softtabstop' to the value of 'shiftwidth'.
NOTE: in most cases, using 'softtabstop' is a better option. Have a look at section 30.5 of the user guide for detailed explanations on how Vim works with tabs and spaces.

'smoothscroll' 'sms' 'nosmoothscroll' 'nosms'
'smoothscroll' 'sms'	boolean (default off)
local to window

Scrolling works with screen lines. When 'wrap' is set and the first line in the window wraps part of it may not be visible, as if it is above the window. "<<<" is displayed at the start of the first line, highlighted with hl-NonText.
You may also want to add "lastline" to the 'display' option to show as much of the last line as possible.
NOTE: partly implemented, doesn't work yet for gj and gk.

'softtabstop' 'sts'
'softtabstop' 'sts'	number (default 0)
local to buffer

Create soft tab stops, separated by 'softtabstop' number of columns.
In Insert mode, pressing the <Tab> key will move the cursor to the next soft tab stop, instead of inserting a literal tab. <BS> behaves similarly in reverse. Vim inserts a minimal mix of tab and space characters to produce the visual effect.

This setting does not affect the display of existing tab characters.

A value of 0 disables this behaviour. A negative value makes Vim use 'shiftwidth'. If you plan to use 'sts' and 'shiftwidth' with different values, you might consider setting 'smarttab'.

The 'L' flag in 'cpoptions' alters tab behavior when 'list' is enabled. See also ins-expandtab and user manual section 30.5 for in-depth explanations.

The value of 'softtabstop' will be ignored if 'varsofttabstop' is set to anything other than an empty string.

'spell' 'nospell'
'spell'	boolean (default off)
local to window

When on spell checking will be done. See spell.
The languages are specified with 'spelllang'.

'spellcapcheck' 'spc'
'spellcapcheck' 'spc'	string (default "[.?!]_[\])'"\t]\+")
local to buffer

Pattern to locate the end of a sentence. The following word will be checked to start with a capital letter. If not then it is highlighted with SpellCap hl-SpellCap (unless the word is also badly spelled).
When this check is not wanted make this option empty.
Only used when 'spell' is set.
Be careful with special characters, see option-backslash about including spaces and backslashes.

To set this option automatically depending on the language, see
set-spc-auto.

'spellfile' 'spf'

'spellfile' 'spf'	string (default "")
local to buffer
Disallowed in modeline. no-modeline-option
Name of the word list file where words are added for the zg and zw
commands. It must end in ".{encoding}.add". You need to include the
path, otherwise the file is placed in the current directory.
The path may include characters from 'isfname', ' ', ',', '@' and ':'.

E765

It may also be a comma-separated list of names. A count before the
zg and zw commands can be used to access each. This allows using
a personal word list file and a project word list file.
When a word is added while this option is empty Nvim will use
(and auto-create) stdpath('data')/site/spell/. For the file name the
first language name that appears in 'spelllang' is used, ignoring the
region.
The resulting ".spl" file will be used for spell checking, it does not
have to appear in 'spelllang'.
Normally one file is used for all regions, but you can add the region
name if you want to. However, it will then only be used when
'spellfile' is set to it, for entries in 'spelllang' only files
without region name will be found.
Environment variables are expanded :set_env.

'spelllang' 'spl'

'spelllang' 'spl'	string (default "en")
local to buffer
A comma-separated list of word list names. When the 'spell' option is
on spellchecking will be done for these languages. Example:

```
set spelllang=en_us,nl,medical
```

This means US English, Dutch and medical words are recognized. Words
that are not recognized will be highlighted.
The word list name must consist of alphanumeric characters, a dash or
an underscore. It should not include a comma or dot. Using a dash is
recommended to separate the two letter language name from a
specification. Thus "en-rare" is used for rare English words.
A region name must come last and have the form "_xx", where "xx" is
the two-letter, lower case region name. You can use more than one
region by listing them: "en_us,en_ca" supports both US and Canadian
English, but not words specific for Australia, New Zealand or Great
Britain. (**Note:** currently en_au and en_nz dictionaries are older than
en_ca, en_gb and en_us).
If the name "cjk" is included East Asian characters are excluded from
spell checking. This is useful when editing text that also has Asian
words.
Note that the "medical" dictionary does not exist, it is just an
example of a longer name.

E757

As a special case the name of a .spl file can be given as-is. The
first "_xx" in the name is removed and used as the region name
(_xx is an underscore, two letters and followed by a non-letter).
This is mainly for testing purposes. You must make sure the correct
encoding is used, Vim doesn't check it.
How the related spell files are found is explained here: spell-load.

If the package-spellfile plugin is active and you use a language name
for which Vim cannot find the .spl file in 'runtimepath' the plugin
will ask you if you want to download the file.

After this option has been set successfully, Vim will source the files
"spell/LANG.vim" in 'runtimepath'. "LANG" is the value of 'spelllang'
up to the first character that is not an ASCII letter or number and
not a dash. Also see set-spc-auto.

'spelloptions' 'spo'

'spelloptions' 'spo'	string (default "")
local to buffer
Disallowed in modeline. no-modeline-option
A comma-separated list of options for spell checking:
camel	When a word is CamelCased, assume "Cased" is a
separate word: every upper-case character in a word
that comes after a lower case character indicates the
start of a new word.
noplainbuffer	Only spellcheck a buffer when 'syntax' is enabled,
or when extmarks are set within the buffer. Only
designated regions of the buffer are spellchecked in
this case.

'spellsuggest' 'sps' E5700

'spellsuggest' 'sps'	string (default "best")
global
Disallowed in modeline. no-modeline-option
Methods used for spelling suggestions. Both for the z= command and
the spellsuggest() function. This is a comma-separated list of
items:

best Internal method that works best for English. Finds changes like "fast" and uses a bit of sound-a-like scoring to improve the ordering.

double Internal method that uses two methods and mixes the results. The first method is "fast", the other method computes how much the suggestion sounds like the bad word. That only works when the language specifies sound folding. Can be slow and doesn't always give better results.

fast Internal method that only checks for simple changes: character inserts/deletes/swaps. Works well for simple typing mistakes.

{number} The maximum number of suggestions listed for z=. Not used for spellsuggest(). The number of suggestions is never more than the value of 'lines' minus two.

timeout:{millisec} Limit the time searching for suggestions to {millisec} milliseconds. Applies to the following methods. When omitted the limit is 5000. When negative there is no limit.

file:{filename} Read file {filename}, which must have two columns, separated by a slash. The first column contains the bad word, the second column the suggested good word. Example:

```
theribal/terrible
```

Use this for common mistakes that do not appear at the top of the suggestion list with the internal methods. Lines without a slash are ignored, use this for comments.
The word in the second column must be correct, otherwise it will not be used. Add the word to an ".add" file if it is currently flagged as a spelling mistake.
The file is used for all languages.

expr:{expr} Evaluate expression {expr}. Use a function to avoid trouble with spaces. Best is to call a function without arguments, see expr-option-function. v:val holds the badly spelled word. The expression must evaluate to a List of Lists, each with a suggestion and a score.
Example:

```
[['the', 33], ['that', 44]]
```

Set 'verbose' and use z= to see the scores that the internal methods use. A lower score is better.
This may invoke spellsuggest() if you temporarily set 'spellsuggest' to exclude the "expr:" part. Errors are silently ignored, unless you set the 'verbose' option to a non-zero value.

Only one of "best", "double" or "fast" may be used. The others may appear several times in any order. Example:

```
set sps=file:~/.config/nvim/sugg,best,expr:MySuggest()
```

Environment variables are expanded :set_env.

'splitbelow' 'sb' 'nosplitbelow' 'nosb'
'splitbelow' 'sb' boolean (default off)
global
When on, splitting a window will put the new window below the current one. :split

'splitkeep' 'spk'
'splitkeep' 'spk' string (default "cursor")
global
The value of this option determines the scroll behavior when opening, closing or resizing horizontal splits.

Possible values are:
cursor Keep the same relative cursor position.
screen Keep the text on the same screen line.
topline Keep the topline the same.

For the "screen" and "topline" values, the cursor position will be changed when necessary. In this case, the jumplist will be populated with the previous cursor position. For "screen", the text cannot always be kept on the same screen line when 'wrap' is enabled.

'splitright' 'spr' 'nosplitright' 'nospr'
'splitright' 'spr' boolean (default off)
global
When on, splitting a window will put the new window right of the current one. :vsplit

'startofline' 'sol' 'nostartofline' 'nosol'
'startofline' 'sol' boolean (default off)
global
When "on" the commands listed below move the cursor to the first non-blank of the line. When off the cursor is kept in the same column (if possible). This applies to the commands:

- CTRL-D, CTRL-U, CTRL-B, CTRL-F, "G", "H", "M", "L", "gg"

- "d", "<<", "==" and ">>" with a linewise operator

  (operator-resulting-pos)
- "%" with a count

- buffer changing commands (CTRL-^, :bnext, :bNext, etc.)

- Ex commands that only have a line number, e.g., ":25" or ":+".

In case of buffer changing commands the cursor is placed at the column where it was the last time the buffer was edited.

'statuscolumn' 'stc'
'statuscolumn' 'stc'	string (default "")
local to window
Disallowed in modeline. no-modeline-option

When non-empty, this option determines the content of the area to the side of a window, normally containing the fold, sign and number columns. The format of this option is like that of 'statusline'.

Some of the items from the 'statusline' format are different for 'statuscolumn':

```
%l      line number column for currently drawn line
%s      sign column for currently drawn line
%C      fold column for currently drawn line
```

The 'statuscolumn' width follows that of the default columns and adapts to the 'numberwidth', 'signcolumn' and 'foldcolumn' option values (regardless of whether the sign and fold items are present). Additionally, the 'statuscolumn' grows with the size of the evaluated format string, up to a point (following the maximum size of the default fold, sign and number columns). Shrinking only happens when the number of lines in a buffer changes, or the 'statuscolumn' option is set.

```
The v:lnum    variable holds the line number to be drawn.
The v:relnum  variable holds the relative line number to be drawn.
The v:virtnum variable is negative when drawing virtual lines, zero
              when drawing the actual buffer line, and positive when
              drawing the wrapped part of a buffer line.
```

When using v:relnum, keep in mind that cursor movement by itself will not cause the 'statuscolumn' to update unless 'relativenumber' is set.

Examples:

```
" Line number with bar separator and click handlers:
set statuscolumn=%@SignCb@%s%=%T%@NumCb@%l│%T

" Line numbers in hexadecimal for non wrapped part of lines:
let &stc='%=%{v:virtnum>0?"":printf("%x",v:lnum)} '

" Human readable line numbers with thousands separator:
let &stc='%{substitute(v:lnum,"\\d\\zs\\ze\\'
          . '%(\\d\\d\\d\\)\\+$",",","g")}'

" Both relative and absolute line numbers with different
" highlighting for odd and even relative numbers:
let &stc='%#NonText#%{&nu?v:lnum:""}' .
 '%=%{&rnu&&(v:lnum%2)?"\ ".v:relnum:""}' .
 '%#LineNr#%{&rnu&&!(v:lnum%2)?"\ ".v:relnum:""}'
```

WARNING: this expression is evaluated for each screen line so defining an expensive expression can negatively affect render performance.

'statusline' 'stl' E540 E542
'statusline' 'stl'	string (default is very long)
global or local to window global-local

Sets the status-line.

Contains printf-style "%" items interspersed with normal text, where each item has the form:

```
%-0{minwid}.{maxwid}{item}
```

All fields except {item} are optional. Use "%%" to show a literal "%" char. Setting to empty (:set statusline=) sets the global value to the default.

stl-%!

When the option starts with "%!" then it is used as an expression, evaluated and the result is used as the option value. Example:

```
set statusline=%!MyStatusLine()
```

The g:statusline_winid variable will be set to the window-ID of the window that the status line belongs to.

The result can contain %{} items that will be evaluated too.
Note that the "%!" expression is evaluated in the context of the
current window and buffer, while %{} items are evaluated in the
context of the window that the statusline belongs to.

When there is an error while evaluating the option it will be reset to
its default value to avoid further errors. Otherwise screen updating
would loop. When the result contains unprintable characters the
result is unpredictable.

When 'ruler' is on, the default 'statusline' includes 'rulerformat'.
See note below.
Note that if 'statusline' is configured without including the ruler,
the only effect of 'ruler' when this option is set (and 'laststatus'
is 2 or 3) is controlling the output of CTRL-G.

field	meaning
-	Left justify the item. The default is right justified when minwid is larger than the length of the item.
0	Leading zeroes in numeric items. Overridden by "-".
minwid	Minimum width of the item, padding as set by "-" & "0". Value must be 50 or less.
maxwid	Maximum width of the item. Truncation occurs with a "<" on the left for text items. Numeric items will be shifted down to maxwid-2 digits followed by ">"number where number is the amount of missing digits, much like an exponential notation.
item	A one letter code as described below.

Following is a description of the possible statusline items. The
second character in "item" is the type:
        "N" for number
        "S" for string
        "F" for flags as described below
        "-" not applicable

item meaning
f S   Path to the file in the buffer, as typed or relative to current
      directory.
F S   Full path to the file in the buffer.
t S   File name (tail) of file in the buffer.
m F   Modified flag, text is "[+]"; "[-]" if 'modifiable' is off.
M F   Modified flag, text is ",+" or ",-".
r F   Readonly flag, text is "[RO]".
R F   Readonly flag, text is ",RO".
h F   Help buffer flag, text is "[help]".
H F   Help buffer flag, text is ",HLP".
w F   Preview window flag, text is "[Preview]".
W F   Preview window flag, text is ",PRV".
y F   Type of file in the buffer, e.g., "[vim]". See 'filetype'.
Y F   Type of file in the buffer, e.g., ",VIM". See 'filetype'.
q S   "[Quickfix List]", "[Location List]" or empty.
k S   Value of "b:keymap_name" or 'keymap' when :lmap mappings are
      being used: "<keymap>"
n N   Buffer number.
b N   Value of character under cursor.
B N   As above, in hexadecimal.
o N   Byte number in file of byte under cursor, first byte is 1.
      Mnemonic: Offset from start of file (with one added)
O N   As above, in hexadecimal.
l N   Line number.
L N   Number of lines in buffer.
c N   Column number (byte index).
v N   Virtual column number (screen column).
V N   Virtual column number as -{num}. Not displayed if equal to 'c'.
p N   Percentage through file in lines as in CTRL-G.
P S   Percentage through file of displayed window. This is like the
      percentage described for 'ruler'. Always 3 in length, unless
      translated.
S S   'showcmd' content, see 'showcmdloc'.
a S   Argument list status as in default title. ({current} of {max})
      Empty if the argument file count is zero or one.
{ NF Evaluate expression between "%{" and "}" and substitute result.
      Note that there is no "%" before the closing "}". The
      expression cannot contain a "}" character, call a function to
      work around that. See stl-%{ below. Use "%0{" to insert the
      result verbatim.
{% - This is almost same as "{" except the result of the expression is
      re-evaluated as a statusline format string. Thus if the
      return value of expr contains "%" items they will get expanded.
      The expression can contain the "}" character, the end of
      expression is denoted by "%}".
      For example:

```
func! Stl_filename() abort
    return "%t"
endfunc
```

        stl=%{Stl_filename()}   results in "%t"
        stl=%{%Stl_filename()%} results in "Name of current file"
      The default status line uses this to include 'rulerformat', see

note below.
%} - End of "{%" expression
(- Start of item group. Can be used for setting the width and alignment of a section. Must be followed by %) somewhere.
) - End of item group. No width fields allowed.
T N For 'tabline': start of tabpage N label. Use %T or %X to end the label. Clicking this label with left mouse button switches to the specified tabpage, while clicking it with middle mouse button closes the specified tabpage.
X N For 'tabline': start of close tab N label. Use %X or %T to end the label, e.g.: %3Xclose%X. Use %999X for a "close current tab" label. Clicking this label with left mouse button closes the specified tabpage.
@ N Start of execute function label. Use %X or %T to end the label, e.g.: %10@SwitchBuffer@foo.c%X. Clicking this label runs the specified function: in the example when clicking once using left mouse button on "foo.c", a SwitchBuffer(10, 1, 'l', ' ') expression will be run. The specified function receives the following arguments in order:

1. minwid field value or zero if no N was specified

2. number of mouse clicks to detect multiple clicks

3. mouse button used: "l", "r" or "m" for left, right or middle

button respectively; one should not rely on third argument being only "l", "r" or "m": any other non-empty string value that contains only ASCII lower case letters may be expected for other mouse buttons

4. modifiers pressed: string which contains "s" if shift

modifier was pressed, "c" for control, "a" for alt and "m" for meta; currently if modifier is not pressed string contains space instead, but one should not rely on presence of spaces or specific order of modifiers: use stridx() to test whether some modifier is present; string is guaranteed to contain only ASCII letters and spaces, one letter per modifier; "?" modifier may also be present, but its presence is a bug that denotes that new mouse button recognition was added without modifying code that reacts on mouse clicks on this label.

Use getmousepos().winid in the specified function to get the corresponding window-ID of the clicked item.
< - Where to truncate line if too long. Default is at the first item. Truncation markers within item groups apply to the truncation of that group until its maxwid is reached. In case of several competing truncation markers, the first wins. No width fields allowed.
= - Separation point between alignment sections. Each section will be separated by an equal number of spaces. With one %= what comes after it will be right-aligned. With two %= there is a middle part, with white space left and right of it. Alignment sections within item groups will be separated until minwid of the group is reached. No width fields allowed.
- Set highlight group. The name must follow and then a # again. Thus use %#HLname# for highlight group HLname. The same highlighting is used, also for the statusline of non-current windows.
$ - Same as #, except the %$HLname$ group will inherit from preceding highlight attributes.
* - Set highlight group to User{N}, where {N} is taken from the minwid field, e.g. %1*. Restore normal highlight with %* or %0*. The difference between User{N} and StatusLine will be applied to StatusLineNC for the statusline of non-current windows. The number N must be between 1 and 9. See hl-User1..9
#(- Start of a highlight scope. The exact highlighting before the scope is reset at the end. No width fields allowed.
#) - End of highlight scope. No width fields allowed.

When displaying a flag, Vim removes the leading comma, if any, when that flag comes right after plaintext. This will make a nice display when flags are used like in the examples below.

When all items in a group becomes an empty string (i.e. flags that are not set) and a minwid is not set for the group, the whole group will become empty. This will make a group like the following disappear completely from the statusline when none of the flags are set.

```
set statusline=...%(\ [%M%R%H]%)...
```

Beware that an expression is evaluated each and every time the status line is displayed.
stl-%{ g:actual_curbuf g:actual_curwin
While evaluating %{} the current buffer and current window will be set

temporarily to that of the window (and buffer) whose statusline is currently being drawn. The expression will evaluate in this context. The variable "g:actual_curbuf" is set to the bufnr() number of the real current buffer and "g:actual_curwin" to the window-ID of the real current window. These values are strings.

The 'statusline' option will be evaluated in the sandbox if set from a modeline, see sandbox-option.
This option cannot be set in a modeline when 'modelineexpr' is off.

It is not allowed to change text or jump to another window while evaluating 'statusline' textlock.

If the statusline is not updated when you want it (e.g., after setting a variable that's used in an expression), you can force an update by using :redrawstatus.

A result of all digits is regarded a number for display purposes. Otherwise the result is taken as flag text and applied to the rules described above.

stl-%0{

With %0{ neither applies: the result is inserted as a literal string.

Watch out for errors in expressions. They may render Vim unusable! If you are stuck, hold down ':' or 'Q' to get a prompt, then quit and edit your vimrc or whatever with "vim --clean" to get it right.

Examples:
A simple version of the standard status line with 'ruler' set

```
set statusline=%f\ %h%w%m%r%=\ %-14.(%l,%c%V%)\ %P
```

Similar, but add ASCII value of char under the cursor (like "ga")

```
set statusline=%f%h%m%r%=\ %b\ 0x%B\ \ %l,%c%V\ %P
```

Display byte count and byte value, modified flag in red.

```
set statusline=%f%=\ [%1*%M%*%n%R%H]\ %-19(%3l,%02c%03V%)%O'%02b'
hi User1 term=inverse,bold cterm=inverse,bold ctermfg=red
```

Display a ,GZ flag if a compressed file is loaded

```
set statusline=...%r%{VarExists('b:gzflag','\ [GZ]')}%h...
```

In the :autocmd's:

```
let b:gzflag = 1
```

And:

```
unlet b:gzflag
```

And define this function:

```
function VarExists(var, val)
    if exists(a:var) | return a:val | else | return '' | endif
endfunction
```

Note: By default, the status line is truncated from the left, and the ruler from the right. But the status line can include the ruler. To ensure that a top-level (i.e. not inside an item group) %< in 'rulerformat' doesn't change the truncation of the status line,

- 'statusline' can be prepended with an explicit %<, which otherwise

  would not be necessary. Example:

  ```
  set statusline=%<%f%=\ %{%&rulerformat%}
  ```

- Or the ruler can be wrapped in an item group, which also collapses

  any top-level %= in 'rulerformat' unless minwid is specified:

  ```
  set statusline=%f%=\ %(%{%&rulerformat%}%)
  ```

'suffixes' 'su'

'suffixes' 'su' string (default ".bak,~,.o,.h,.info,.swp,.obj")
global
Files with these suffixes get a lower priority when multiple files match a wildcard. See suffixes. Commas can be used to separate the suffixes. Spaces after the comma are ignored. A dot is also seen as the start of a suffix. To avoid a dot or comma being recognized as a separator, precede it with a backslash (see option-backslash about including spaces and backslashes).
See 'wildignore' for completely ignoring files.
The use of :set+= and :set-= is preferred when adding or removing suffixes from the list. This avoids problems when a future version uses another default.

'suffixesadd' 'sua'

'suffixesadd' 'sua'	string (default "")
		local to buffer
	Comma-separated list of suffixes, which are used when searching for a
	file for the "gf", "[I", etc. commands. Example:

```
set suffixesadd=.java
```

'swapfile' 'swf' 'noswapfile' 'noswf'

'swapfile' 'swf'	boolean (default on)
		local to buffer
	Use a swap-file for the buffer (if 'updatecount' is non-zero). The
	'directory' option decides where swapfiles are stored.

	To open a new buffer without creating a swapfile, use :noswapfile.
	To disable for an existing buffer, reset its 'swapfile' option.
	Careful:

- Recovery will be impossible!

- The entire file will be in memory.

	When reset, the swapfile for the current buffer is immediately
	deleted. When re-enabled (and 'updatecount' is non-zero), a swapfile
	is immediately created.

	Used with 'bufhidden' and 'buftype' to specify special-buffers.

'switchbuf' 'swb'

'switchbuf' 'swb'	string (default "uselast")
		global
	This option controls the behavior when switching between buffers.
	This option is checked, when

- jumping to errors with the quickfix commands (:cc, :cn, :cp,

	  etc.).

- jumping to a tag using the :stag command.

- opening a file using the CTRL-W_f or CTRL-W_F command.

- jumping to a buffer using a buffer split command (e.g. :sbuffer,

	  :sbnext, or :sbrewind).
	Possible values (comma-separated list):
	   useopen	If included, jump to the first open window in the
			current tabpage that contains the specified buffer
			(if there is one). Otherwise: Do not examine other
			windows.
	   usetab	Like "useopen", but also consider windows in other
			tabpages.
	   split	If included, split the current window before loading
			a buffer for a quickfix command that display errors.
			Otherwise: do not split, use current window (when used
			in the quickfix window: the previously used window or
			split if there is no other window).
	   vsplit	Just like "split" but split vertically.
	   newtab	Like "split", but open a new tabpage. Overrules
			"split" when both are present.
	   uselast	If included, jump to the previously used window when
			jumping to errors with quickfix commands.
	If a window has 'winfixbuf' enabled, 'switchbuf' is currently not
	applied to the split window.

'synmaxcol' 'smc'

'synmaxcol' 'smc'	number (default 3000)
		local to buffer
	Maximum column in which to search for syntax items. In long lines the
	text after this column is not highlighted and following lines may not
	be highlighted correctly, because the syntax state is cleared.
	This helps to avoid very slow redrawing for an XML file that is one
	long line.
	Set to zero to remove the limit.

'syntax' 'syn'

'syntax' 'syn'	string (default "")
		local to buffer local-noglobal
	When this option is set, the syntax with this name is loaded, unless
	syntax highlighting has been switched off with ":syntax off".
	Otherwise this option does not always reflect the current syntax (the
	b:current_syntax variable does).
	This option is most useful in a modeline, for a file which syntax is
	not automatically recognized. Example, in an IDL file:

```
/* vim: set syntax=idl : */
```

	When a dot appears in the value then this separates two filetype
	names. Example:

```
/* vim: set syntax=c.doxygen : */
```

This will use the "c" syntax first, then the "doxygen" syntax.
Note that the second one must be prepared to be loaded as an addition,
otherwise it will be skipped. More than one dot may appear.
To switch off syntax highlighting for the current file, use:

```
set syntax=OFF
```

To switch syntax highlighting on according to the current value of the
'filetype' option:

```
set syntax=ON
```

What actually happens when setting the 'syntax' option is that the
Syntax autocommand event is triggered with the value as argument.
This option is not copied to another buffer, independent of the 's' or
'S' flag in 'cpoptions'.
Only alphanumeric characters, '.', '-' and '_' can be used.

'tabclose' 'tcl'
'tabclose' 'tcl'	string (default "")
global
This option controls the behavior when closing tabpages (e.g., using
:tabclose). When empty Vim goes to the next (right) tabpage.

Possible values (comma-separated list):
left	If included, go to the previous tabpage instead of
the next one.
uselast	If included, go to the previously used tabpage if
possible. This option takes precedence over the
others.

'tabline' 'tal'
'tabline' 'tal'	string (default "")
global
When non-empty, this option determines the content of the tabpages
line at the top of the Vim window. When empty Vim will use a default
tabpages line. See setting-tabline for more info.

The tabpages line only appears as specified with the 'showtabline'
option and only when there is no GUI tab line. When 'e' is in
'guioptions' and the GUI supports a tab line 'guitablabel' is used
instead. Note that the two tabpages lines are very different.

The value is evaluated like with 'statusline'. You can use
tabpagenr(), tabpagewinnr() and tabpagebuflist() to figure out
the text to be displayed. Use "%1T" for the first label, "%2T" for
the second one, etc. Use "%X" items for closing labels.

When changing something that is used in 'tabline' that does not
trigger it to be updated, use :redrawtabline.
This option cannot be set in a modeline when 'modelineexpr' is off.

Keep in mind that only one of the tabpages is the current one, others
are invisible and you can't jump to their windows.

'tabpagemax' 'tpm'
'tabpagemax' 'tpm'	number (default 50)
global
Maximum number of tabpages to be opened by the -p command line
argument or the ":tab all" command. tabpage

'tabstop' 'ts'
'tabstop' 'ts'	number (default 8)
local to buffer
Defines the column multiple used to display the Horizontal Tab
character (ASCII 9); a Horizontal Tab always advances to the next tab
stop.
The value must be at least 1 and at most 9999.
If 'vartabstop' is set, this option is ignored.
Leave it at 8 unless you have a strong reason (see usr 30.5).

'tagbsearch' 'tbs' 'notagbsearch' 'notbs'
'tagbsearch' 'tbs'	boolean (default on)
global
When searching for a tag (e.g., for the :ta command), Vim can either
use a binary search or a linear search in a tags file. Binary
searching makes searching for a tag a LOT faster, but a linear search
will find more tags if the tags file wasn't properly sorted.
Vim normally assumes that your tags files are sorted, or indicate that
they are not sorted. Only when this is not the case does the
'tagbsearch' option need to be switched off.

When 'tagbsearch' is on, binary searching is first used in the tags
files. In certain situations, Vim will do a linear search instead for
certain files, or retry all files with a linear search. When
'tagbsearch' is off, only a linear search is done.

Linear searching is done anyway, for one file, when Vim finds a line
at the start of the file indicating that it's not sorted:

```
!_TAG_FILE_SORTED	0	/some comment/
```

[The whitespace before and after the '0' must be a single <Tab>]

When a binary search was done and no match was found in any of the files listed in 'tags', and case is ignored or a pattern is used instead of a normal tag name, a retry is done with a linear search. Tags in unsorted tags files, and matches with different case will only be found in the retry.

If a tag file indicates that it is case-fold sorted, the second, linear search can be avoided when case is ignored. Use a value of '2' in the "!_TAG_FILE_SORTED" line for this. A tag file can be case-fold sorted with the -f switch to "sort" in most unices, as in the command: "sort -f -o tags tags". For Universal ctags and Exuberant ctags version 5.x or higher (at least 5.5) the --sort=foldcase switch can be used for this as well. Note that case must be folded to uppercase for this to work.

By default, tag searches are case-sensitive. Case is ignored when 'ignorecase' is set and 'tagcase' is "followic", or when 'tagcase' is "ignore".
Also when 'tagcase' is "followscs" and 'smartcase' is set, or 'tagcase' is "smart", and the pattern contains only lowercase characters.

When 'tagbsearch' is off, tags searching is slower when a full match exists, but faster when no full match exists. Tags in unsorted tags files may only be found with 'tagbsearch' off.
When the tags file is not sorted, or sorted in a wrong way (not on ASCII byte value), 'tagbsearch' should be off, or the line given above must be included in the tags file.
This option doesn't affect commands that find all matching tags (e.g., command-line completion and ":help").

'tagcase' 'tc'

'tagcase' 'tc' string (default "followic")
global or local to buffer global-local

This option specifies how case is handled when searching the tags file:

followic	Follow the 'ignorecase' option
followscs	Follow the 'smartcase' and 'ignorecase' options
ignore	Ignore case
match	Match case
smart	Ignore case unless an upper case letter is used

'tagfunc' 'tfu'

'tagfunc' 'tfu' string (default "")
local to buffer
Disallowed in modeline. no-modeline-option

This option specifies a function to be used to perform tag searches (including taglist()).
The function gets the tag pattern and should return a List of matching tags. See tag-function for an explanation of how to write the function and an example. The value can be the name of a function, a lambda or a Funcref. See option-value-function for more information.

'taglength' 'tl'

'taglength' 'tl' number (default 0)
global

If non-zero, tags are significant up to this number of characters.

'tagrelative' 'tr' 'notagrelative' 'notr'

'tagrelative' 'tr' boolean (default on)
global

If on and using a tags file in another directory, file names in that tags file are relative to the directory where the tags file is.

'tags' 'tag' E433

'tags' 'tag' string (default "./tags;,tags")
global or local to buffer global-local

Filenames for the tag command, separated by spaces or commas. To include a space or comma in a file name, precede it with backslashes (see option-backslash about including spaces/commas and backslashes). When a file name starts with "./", the '.' is replaced with the path of the current file. But only when the 'd' flag is not included in 'cpoptions'. Environment variables are expanded :set_env. Also see tags-option.
"*", "**" and other wildcards can be used to search for tags files in a directory tree. See file-searching. E.g., "/lib/**/tags" will find all files named "tags" below "/lib". The filename itself cannot contain wildcards, it is used as-is. E.g., "/lib/**/tags?" will find files called "tags?".
The tagfiles() function can be used to get a list of the file names actually used.
The use of :set+= and :set-= is preferred when adding or removing file names from the list. This avoids problems when a future version uses another default.

'tagstack' 'tgst' 'notagstack' 'notgst'

'tagstack' 'tgst' boolean (default on)
global

When on, the tagstack is used normally. When off, a ":tag" or ":tselect" command with an argument will not push the tag onto the tagstack. A following ":tag" without an argument, a ":pop" command or any other command that uses the tagstack will use the unmodified tagstack, but does change the pointer to the active entry.

Resetting this option is useful when using a ":tag" command in a
mapping which should not change the tagstack.

'termbidi' 'tbidi' 'notermbidi' 'notbidi'
'termbidi' 'tbidi' boolean (default off)
global
The terminal is in charge of Bi-directionality of text (as specified
by Unicode). The terminal is also expected to do the required shaping
that some languages (such as Arabic) require.
Setting this option implies that 'rightleft' will not be set when
'arabic' is set and the value of 'arabicshape' will be ignored.
Note that setting 'termbidi' has the immediate effect that
'arabicshape' is ignored, but 'rightleft' isn't changed automatically.
For further details see l10n-arabic.txt.

'termguicolors' 'tgc' 'notermguicolors' 'notgc'
'termguicolors' 'tgc' boolean (default off)
global
Enables 24-bit RGB color in the TUI. Uses "gui" :highlight
attributes instead of "cterm" attributes. guifg
Requires an ISO-8613-3 compatible terminal.

Nvim will automatically attempt to determine if the host terminal
supports 24-bit color and will enable this option if it does
(unless explicitly disabled by the user).

'termpastefilter' 'tpf'
'termpastefilter' 'tpf' string (default "BS,HT,ESC,DEL")
global
A comma-separated list of options for specifying control characters
to be removed from the text pasted into the terminal window. The
supported values are:

BS Backspace

HT TAB

FF Form feed

ESC Escape

DEL DEL

C0 Other control characters, excluding Line feed and
Carriage return < ' '

C1 Control characters 0x80...0x9F

'termsync' 'notermsync'
'termsync' boolean (default on)
global
If the host terminal supports it, buffer all screen updates
made during a redraw cycle so that each screen is displayed in
the terminal all at once. This can prevent tearing or flickering
when the terminal updates faster than Nvim can redraw.

'textwidth' 'tw'
'textwidth' 'tw' number (default 0)
local to buffer
Maximum width of text that is being inserted. A longer line will be
broken after white space to get this width. A zero value disables
this.
When 'textwidth' is zero, 'wrapmargin' may be used. See also
'formatoptions' and ins-textwidth.
When 'formatexpr' is set it will be used to break the line.

'thesaurus' 'tsr'
'thesaurus' 'tsr' string (default "")
global or local to buffer global-local
List of file names, separated by commas, that are used to lookup words
for thesaurus completion commands i_CTRL-X_CTRL-T. See
compl-thesaurus.

This option is not used if 'thesaurusfunc' is set, either for the
buffer or globally.

To include a comma in a file name precede it with a backslash. Spaces
after a comma are ignored, otherwise spaces are included in the file
name. See option-backslash about using backslashes. The use of
:set+= and :set-= is preferred when adding or removing directories
from the list. This avoids problems when a future version uses
another default.
Environment variables are expanded :set_env.
Backticks cannot be used in this option for security reasons.

'thesaurusfunc' 'tsrfu'
'thesaurusfunc' 'tsrfu' string (default "")
global or local to buffer global-local
Disallowed in modeline. no-modeline-option
This option specifies a function to be used for thesaurus completion
with CTRL-X CTRL-T. i_CTRL-X_CTRL-T See compl-thesaurusfunc.
The value can be the name of a function, a lambda or a Funcref.
See option-value-function for more information.

'tildeop' 'top' 'notildeop' 'notop'
'tildeop' 'top' boolean (default off)
global
When on: The tilde command "~" behaves like an operator.

'timeout' 'to' 'notimeout' 'noto'
'timeout' 'to' boolean (default on)

global
This option and 'timeoutlen' determine the behavior when part of a mapped key sequence has been received. For example, if <c-f> is pressed and 'timeout' is set, Nvim will wait 'timeoutlen' milliseconds for any key that can follow <c-f> in a mapping.

'timeoutlen' 'tm'

'timeoutlen' 'tm' number (default 1000)
global
Time in milliseconds to wait for a mapped sequence to complete.

'title' 'notitle'

'title' boolean (default off)
global
If enabled, Nvim will update the (GUI or terminal) window title. The format is configured by 'titlestring'. By default it looks like:

```
filename [+=-] (path) - Nvim
```

where:

```
filename        the name of the file being edited
-               indicates the file cannot be modified, 'ma' off
+               indicates the file was modified
=               indicates the file is read-only
=+              indicates the file is read-only and modified
(path)          is the path of the file being edited
- Nvim          the server name |v:servername| or "Nvim"
```

'titlelen'

'titlelen' number (default 85)
global
Gives the percentage of 'columns' to use for the length of the window title. When the title is longer, only the end of the path name is shown. A '<' character before the path name is used to indicate this. Using a percentage makes this adapt to the width of the window. But it won't work perfectly, because the actual number of characters available also depends on the font used and other things in the title bar. When 'titlelen' is zero the full path is used. Otherwise, values from 1 to 30000 percent can be used.
'titlelen' is also used for the 'titlestring' option.

'titleold'

'titleold' string (default "")
global
Disallowed in modeline. no-modeline-option
If not empty, this option will be used to set the window title when exiting. Only if 'title' is enabled.

'titlestring'

'titlestring' string (default "")
global
Formats the window title, enabled by the 'title' option.

Contains printf-style "%" items, expanded according to the rules of 'statusline'. If a "%" format is invalid, it is used as-is and no error will be given.

The default (empty) behaviour is equivalent to:

```
set titlestring=%t%(\ %M%)%(\ \(%{expand('%:p:~:h')}\)%)%a\ -\ Nvim
```

Example:

```
auto BufEnter * let &titlestring = hostname() .. "/" .. expand("%:p")
set title titlestring=%<%F%=%l/%L-%P titlelen=70
```

The value of 'titlelen' is used to align items in the middle or right of the available space.

Example: to have the file name first:

```
set titlestring=%t%(\ %M%)%(\ (%{expand(\"%:~:.:h\")})%)%(\ %a%)
```

Note the use of "%{ }" and an expression to get the path of the file, without the file name. The "%(%)" constructs add a separating space only when needed.

NOTE: Use of special characters in 'titlestring' may cause the display to be garbled (e.g., when it contains a CR or NL character).

This option cannot be set in a modeline when 'modelineexpr' is off.

'ttimeout' 'nottimeout'

'ttimeout' boolean (default on)
global
This option and 'ttimeoutlen' determine the behavior when part of a key code sequence has been received by the TUI.

For example if <Esc> (the \x1b byte) is received and 'ttimeout' is set, Nvim waits 'ttimeoutlen' milliseconds for the terminal to complete a key code sequence. If no input arrives before the timeout, a single <Esc> is assumed. Many TUI cursor key codes start with <Esc>.

On very slow systems this may fail, causing cursor keys not to work
sometimes. If you discover this problem you can ":set ttimeoutlen=9999".
Nvim will wait for the next character to arrive after an <Esc>.

'ttimeoutlen' 'ttm'
'ttimeoutlen' 'ttm'	number (default 50)
global
Time in milliseconds to wait for a key code sequence to complete. Also
used for CTRL-\ CTRL-N and CTRL-\ CTRL-G when part of a command has
been typed.

'ttyfast' 'tf' 'nottyfast' 'notf' E1568 $NVIM_NOTTYFAST
'ttyfast' 'tf'	boolean (default on)
global
Enables Nvim TUI features which assume a fast (usually local) host
terminal. During startup, Nvim queries the terminal (for 'background'
detection, etc.) and must wait for a response (or timeout).

If your terminal environment is slow (e.g. remote SSH), or broken
(doesn't respond to queries), Nvim startup may be slower. Therefore
you can disable this option by setting the $NVIM_NOTTYFAST
environment variable before starting Nvim:

```
NVIM_NOTTYFAST=1 nvim
```

The queries are performed early, before --cmd and user config, so
:set nottyfast in your config happens too late.

'undodir' 'udir' E5003
'undodir' 'udir'	string (default "$XDG_STATE_HOME/nvim/undo//")
global
Disallowed in modeline. no-modeline-option
List of directory names for undo files, separated with commas.
See 'backupdir' for details of the format.
"." means using the directory of the file. The undo file name for
"file.txt" is ".file.txt.un~".
For other directories the file name is the full path of the edited
file, with path separators replaced with "%".
When writing: The first directory that exists is used. "." always
works, no directories after "." will be used for writing. If none of
the directories exist Nvim will attempt to create the last directory in
the list.
When reading all entries are tried to find an undo file. The first
undo file that exists is used. When it cannot be read an error is
given, no further entry is used.
See undo-persistence.
Environment variables are expanded :set_env.

Note that unlike 'directory' and 'backupdir', 'undodir' always acts as
though the trailing slashes are present (see 'backupdir' for what this
means).

'undofile' 'udf' 'noundofile' 'noudf'
'undofile' 'udf'	boolean (default off)
local to buffer
When on, Vim automatically saves undo history to an undo file when
writing a buffer to a file, and restores undo history from the same
file on buffer read.
The directory where the undo file is stored is specified by 'undodir'.
For more information about this feature see undo-persistence.
The undo file is not read when 'undoreload' causes the buffer from
before a reload to be saved for undo.
When 'undofile' is turned off the undo file is NOT deleted.

'undolevels' 'ul'
'undolevels' 'ul'	number (default 1000)
global or local to buffer global-local
Maximum number of changes that can be undone. Since undo information
is kept in memory, higher numbers will cause more memory to be used.
Nevertheless, a single change can already use a large amount of
memory. Set to 0 for Vi compatibility: One level of undo and "u"
undoes itself:

```
set ul=0
```

But you can also get Vi compatibility by including the 'u' flag in
'cpoptions', and still be able to use CTRL-R to repeat undo.
Also see undo-two-ways.
Set to -1 for no undo at all. You might want to do this only for the
current buffer:

```
setlocal ul=-1
```

This helps when you run out of memory for a single change.

The local value is set to -123456 when the global value is to be used.

Also see clear-undo.

'undoreload' 'ur'
'undoreload' 'ur'	number (default 10000)
global
Save the whole buffer for undo when reloading it. This applies to the
":e!" command and reloading for when the buffer changed outside of
Vim. FileChangedShell

The save only happens when this option is negative or when the number of lines is smaller than the value of this option.
Set this option to zero to disable undo for a reload.

When saving undo for a reload, any undo file is not read.

Note that this causes the whole buffer to be stored in memory. Set this option to a lower value if you run out of memory.

'updatecount' 'uc'
'updatecount' 'uc' number (default 200)
global
The swap-file will be written after typing this many characters.

- Ignored in buffers where 'buftype' is "nofile" or "nowrite".

- Initialized to 10000 when editing in readonly -R mode.

- To disable swapfiles per-buffer, unset the 'swapfile' option.

- To disable swapfiles globally, set this option to zero (or start

with -n). See crash-recovery. Existing swapfiles are not deleted.
- When re-enabled (from zero to non-zero), swapfiles are created for

all buffers that have 'swapfile' set.

'updatetime' 'ut'
'updatetime' 'ut' number (default 4000)
global
If this many milliseconds nothing is typed the swap file will be written to disk (see crash-recovery). Also used for the CursorHold autocommand event.

'varsofttabstop' 'vsts'
'varsofttabstop' 'vsts' string (default "")
local to buffer
Defines variable-width soft tab stops. The value is a comma-separated list of widths in columns. Each width defines the number of columns before the next soft tab stop. The last value repeats indefinitely.

For example, when editing assembly language files where statements start in the 9th column and comments in the 41st, it may be useful to use the following:

```
set varsofttabstop=8,32,8
```

This sets soft tab stops at column 8, then at column 40 (8 + 32), and every 8 columns thereafter.

Note: this setting overrides 'softtabstop'.
See section 30.5 of the user manual for detailed explanations on how Vim works with tabs and spaces.

'vartabstop' 'vts'
'vartabstop' 'vts' string (default "")
local to buffer
Defines variable-width tab stops. The value is a comma-separated list of widths in columns. Each width defines the number of columns before the next tab stop; the last value repeats indefinitely.

For example:

```
:set vartabstop=4,8
```

This places the first tab stop 4 columns from the start of the line and each subsequent tab stop 8 columns apart.

Note: this setting overrides 'tabstop'.
On UNIX, it is recommended to keep the default tabstop value of 8.
Consider setting 'varsofttabstop' instead.
See section 30.5 of the user manual for detailed explanations on how Vim works with tabs and spaces.

'verbose' 'vbs'
'verbose' 'vbs' number (default 0)
global
Sets the verbosity level. Also set by -V and :verbose.

Tracing of assignments to options, mappings, etc. in Lua scripts is enabled at level 1; Lua scripts are not traced when 'verbose' is 0, for performance.

If greater than or equal to a given level, Nvim produces the following messages:

Level	Messages
1	Enables Lua tracing (see above). Does not produce messages.
2	When a file is ":source"'ed, or shada file is read or written.
3	UI info, terminal capabilities.
4	Shell commands.
5	Every searched tags file and include file.

```
        8       Files for which a group of autocommands is executed.
        9       Executed autocommands.
        11      Finding items in a path.
        12      Vimscript function calls.
        13      When an exception is thrown, caught, finished, or discarded.
        14      Anything pending in a ":finally" clause.
        15      Ex commands from a script (truncated at 200 characters).
        16      Ex commands.
```

If 'verbosefile' is set then the verbose messages are not displayed.

'verbosefile' 'vfile'

'verbosefile' 'vfile' string (default "")
                       global
                       Disallowed in modeline. no-modeline-option
When not empty all messages are written in a file with this name.
When the file exists messages are appended.
Writing to the file ends when Vim exits or when 'verbosefile' is made
empty. Writes are buffered, thus may not show up for some time.
Setting 'verbosefile' to a new value is like making it empty first.
The difference with :redir is that verbose messages are not
displayed when 'verbosefile' is set.
Environment variables are expanded :set_env.

'viewdir' 'vdir'

'viewdir' 'vdir'   string (default "$XDG_STATE_HOME/nvim/view//")
                       global
                       Disallowed in modeline. no-modeline-option
Name of the directory where to store files for :mkview.
Environment variables are expanded :set_env.

'viewoptions' 'vop'

'viewoptions' 'vop'   string (default "folds,cursor,curdir")
                       global
Changes the effect of the :mkview command. It is a comma-separated
list of words. Each word enables saving and restoring something:
word   **save and restore**
   cursor   cursor position in file and in window
   curdir   local current directory, if set with :lcd
   folds   manually created folds, opened/closed folds and local
               fold options
   options   options and mappings local to a window or buffer (not
               global values for local options), except Lua functions
               option-value-function.
   localoptions same as "options"
   slash   deprecated Always enabled. Uses "/" in filenames.
   unix   deprecated Always enabled. Uses "\n" line endings.

'virtualedit' 've'

'virtualedit' 've'   string (default "")
                       global or local to window global-local
A comma-separated list of these words:
    block   Allow virtual editing in Visual block mode.
    insert   Allow virtual editing in Insert mode.
    all   Allow virtual editing in all modes.
    onemore   Allow the cursor to move just past the end of the line
    none   When used as the local value, do not allow virtual
               editing even when the global value is set. When used
               as the global value, "none" is the same as "".
    NONE   Alternative spelling of "none".

Virtual editing means that the cursor can be positioned where there is
no actual character. This can be halfway into a tab or beyond the end
of the line. Useful for selecting a rectangle in Visual mode and
editing a table.
"onemore" is not the same, it will only allow moving the cursor just
after the last character of the line. This makes some commands more
consistent. Previously the cursor was always past the end of the line
if the line was empty. But it is far from Vi compatible. It may also
break some plugins or Vim scripts. For example because l can move
the cursor after the last character. Use with care!
Using the $ command will move to the last character in the line, not
past it. This may actually move the cursor to the left!
The g$ command will move to the end of the screen line.
It doesn't make sense to combine "all" with "onemore", but you will
not get a warning for it.
When combined with other words, "none" is ignored.

'visualbell' 'vb' 'novisualbell' 'novb'

'visualbell' 'vb'   boolean (default off)
                       global
Use visual bell instead of beeping. Also see 'errorbells'.

'warn' 'nowarn'

'warn'   boolean (default on)
                       global
Give a warning message when a shell command is used while the buffer
has been changed.

'whichwrap' 'ww'

'whichwrap' 'ww'   string (default "b,s")
                       global
Allow specified keys that move the cursor left/right to move to the
previous/next line when the cursor is on the first/last character in
the line. Concatenate characters to allow this for these keys:
```
```

```
char   key        mode
                b    <BS>       Normal and Visual
                s    <Space>    Normal and Visual
                h    "h"        Normal and Visual (not recommended)
                l    "l"        Normal and Visual (not recommended)
                <    <Left>     Normal and Visual
                >    <Right>    Normal and Visual
                ~    "~"        Normal
                [    <Left>     Insert and Replace
                ]    <Right>    Insert and Replace
        For example:
```

```
set ww=<,>,[,]
```

```
        allows wrap only when cursor keys are used.
        When the movement keys are used in combination with a delete or change
        operator, the <EOL> also counts for a character.  This makes "3h"
        different from "3dh" when the cursor crosses the end of a line.  This
        is also true for "x" and "X", because they do the same as "dl" and
        "dh".  If you use this, you may also want to use the mapping
        ":map <BS> X" to make backspace delete the character in front of the
        cursor.
        When 'l' is included and it is used after an operator at the end of a
        line (not an empty line) then it will not move to the next line.  This
        makes "dl", "cl", "yl" etc. work normally.

                                                'wildchar' 'wc'
'wildchar' 'wc'         number  (default <Tab>)
                        global
        Character you have to type to start wildcard expansion in the
        command-line, as specified with 'wildmode'.
        More info here: cmdline-completion.
        The character is not recognized when used inside a macro.  See
        'wildcharm' for that.
        Some keys will not work, such as CTRL-C, <CR> and Enter.
        <Esc> can be used, but hitting it twice in a row will still exit
        command-line as a failsafe measure.
        Although 'wc' is a number option, it can be specified as a number, a
        single character, a key-notation (e.g. <Up>, <C-F>) or a letter
        preceded with a caret (e.g. ^F is CTRL-F):
```

```
:set wc=27
:set wc=X
:set wc=^I
set wc=<Tab>
```

```
        'wildchar' also enables completion in search pattern contexts such as
        /, ?, :s, :g, :v, and :vim.  To insert a literal <Tab>
        instead of triggering completion, type <C-V><Tab> or "\t".
        See also 'wildoptions' and wildtrigger().

                                                'wildcharm' 'wcm'
'wildcharm' 'wcm'       number  (default 0)
                        global
        'wildcharm' works exactly like 'wildchar', except that it is
        recognized when used inside a macro.  You can find "spare"
        command-line keys suitable for this option by looking at
        ex-edit-index.  Normally you'll never actually type 'wildcharm',
        just use it in mappings that automatically invoke completion mode,
        e.g.:
```

```
set wcm=<C-Z>
cnoremap ss so $vim/sessions/*.vim<C-Z>
```

```
        Then after typing :ss you can use CTRL-P & CTRL-N.

                                                'wildignore' 'wig'
'wildignore' 'wig'      string  (default "")
                        global
        A list of file patterns.  A file that matches with one of these
        patterns is ignored when expanding wildcards, completing file or
        directory names, and influences the result of expand(), glob() and
        globpath() unless a flag is passed to disable this.
        The pattern is used like with :autocmd, see autocmd-pattern.
        Also see 'suffixes'.
        Example:
```

```
set wildignore=*.o,*.obj
```

```
        The use of :set+= and :set-= is preferred when adding or removing
        a pattern from the list.  This avoids problems when a future version
        uses another default.

                'wildignorecase' 'wic' 'nowildignorecase' 'nowic'
'wildignorecase' 'wic'  boolean (default off)
                        global
        When set case is ignored when completing file names and directories.
        Has no effect when 'fileignorecase' is set.
        Does not apply when the shell is used to expand wildcards, which
        happens when there are special characters.
```

'wildmenu' 'wmnu' 'nowildmenu' 'nowmnu'
'wildmenu' 'wmnu'	boolean (default on)
		global
	When 'wildmenu' is on, command-line completion operates in an enhanced
	mode. On pressing 'wildchar' (usually <Tab>) to invoke completion,
	the possible matches are shown.
	When 'wildoptions' contains "pum", then the completion matches are
	shown in a popup menu. Otherwise they are displayed just above the
	command line, with the first match highlighted (overwriting the status
	line, if there is one).
	Keys that show the previous/next match, such as <Tab> or
	CTRL-P/CTRL-N, cause the highlight to move to the appropriate match.
	'wildmode' must specify "full": "longest" and "list" do not start
	'wildmenu' mode. You can check the current mode with wildmenumode().
	The menu is cancelled when a key is hit that is not used for selecting
	a completion.

	While the menu is active these keys have special meanings:
	CTRL-P		- go to the previous entry
	CTRL-N		- go to the next entry
	<Left> <Right>	- select previous/next match (like CTRL-P/CTRL-N)
	<PageUp>	- select a match several entries back
	<PageDown>	- select a match several entries further
	<Up>		- in filename/menu name completion: move up into
			  parent directory or parent menu.
	<Down>		- in filename/menu name completion: move into a
			  subdirectory or submenu.
	<CR>		- in menu completion, when the cursor is just after a
			  dot: move into a submenu.
	CTRL-E		- end completion, go back to what was there before
			  selecting a match.
	CTRL-Y		- accept the currently selected match and stop
			  completion.

	If you want <Left> and <Right> to move the cursor instead of selecting
	a different match, use this:

```
cnoremap <Left> <Space><BS><Left>
cnoremap <Right> <Space><BS><Right>
```

	hl-WildMenu highlights the current match.

'wildmode' 'wim'
'wildmode' 'wim'	string (default "full")
		global
	Completion mode used for the character specified with 'wildchar'.
	This option is a comma-separated list of up to four parts,
	corresponding to the first, second, third, and fourth presses of
	'wildchar'. Each part is a colon-separated list of completion
	behaviors, which are applied simultaneously during that phase.

	The possible behavior values are:
	""		Only complete (insert) the first match. No further
			matches are cycled or listed.
	"full"		Complete the next full match. Cycles through all
			matches, returning to the original input after the
			last match. If 'wildmenu' is enabled, it will be
			shown.
	"longest"	Complete to the longest common substring. If this
			doesn't extend the input, the next 'wildmode' part is
			used.
	"list"		If multiple matches are found, list all of them.
	"lastused"	When completing buffer names, sort them by most
			recently used (excluding the current buffer). Only
			applies to buffer name completion.
	"noselect"	If 'wildmenu' is enabled, show the menu but do not
			preselect the first item.
	"noinsert"	If 'wildmenu' is enabled, show the menu and preselect
			the first match, but do not insert it in the command
			line. If both "noinsert" and "noselect" are present,
			"noselect" takes precedence.
	If only one match exists, it is completed fully, unless "noselect" or
	"noinsert" is specified.

	Some useful combinations of colon-separated values:
	"longest:full"		Start with the longest common string and show
				'wildmenu' (if enabled). Does not cycle
				through full matches.
	"list:full"		List all matches and complete first match.
	"list:longest"		List all matches and complete till the longest
				common prefix.
	"list:lastused"		List all matches. When completing buffers,
				sort them by most recently used (excluding the
				current buffer).
	"noselect:lastused"	Do not preselect the first item in 'wildmenu'
				if it is active. When completing buffers,
				sort them by most recently used (excluding the
				current buffer).

	Examples:

```
set wildmode=full
```

Complete full match on every press (default behavior)

```
set wildmode=longest,full
```

First press: longest common substring
Second press: cycle through full matches

```
set wildmode=list:full
```

First press: list all matches and complete the first one

```
set wildmode=list,full
```

First press: list matches only
Second press: complete full matches

```
set wildmode=longest,list
```

First press: longest common substring
Second press: list all matches

```
set wildmode=noselect:full
```

First press: show 'wildmenu' without completing or selecting
Second press: cycle full matches

```
set wildmode=noselect:lastused,full
```

Same as above, but buffer matches are sorted by time last used
More info here: cmdline-completion.

'wildoptions' 'wop'
'wildoptions' 'wop'	string (default "pum,tagfile")
global
A list of words that change how cmdline-completion is done.
The following values are supported:
  exacttext	When this flag is present, search pattern completion
(e.g., in /, ?, :s, :g, :v, and :vim)
shows exact buffer text as menu items, without
preserving regex artifacts like position
anchors (e.g., /\<). This provides more intuitive
menu items that match the actual buffer text.
However, searches may be less accurate since the
pattern is not preserved exactly.
By default, Vim preserves the typed pattern (with
anchors) and appends the matched word. This preserves
search correctness, especially when using regular
expressions or with 'smartcase' enabled. However, the
case of the appended matched word may not exactly
match the case of the word in the buffer.
  fuzzy	Use fuzzy-matching to find completion matches. When
this value is specified, wildcard expansion will not
be used for completion. The matches will be sorted by
the "best match" rather than alphabetically sorted.
This will find more matches than the wildcard
expansion. Currently fuzzy matching based completion
is not supported for file and directory names and
instead wildcard expansion is used.
  pum	Display the completion matches using the popup menu in
the same style as the ins-completion-menu. When an
info popup is shown next to the menu, it can be
scrolled by moving the mouse pointer on top of it and
using the scroll wheel.
  tagfile	When using CTRL-D to list matching tags, the kind of
tag and the file of the tag is listed. Only one match
is displayed per line. Often used tag kinds are:
d	#define
f	function

This option does not apply to ins-completion. See 'completeopt' for
that.

'winaltkeys' 'wak'
'winaltkeys' 'wak'	string (default "menu")
global
only used in Win32
Some GUI versions allow the access to menu entries by using the ALT
key in combination with a character that appears underlined in the
menu. This conflicts with the use of the ALT key for mappings and
entering special characters. This option tells what to do:
  no	Don't use ALT keys for menus. ALT key combinations can be
mapped, but there is no automatic handling.
  yes	ALT key handling is done by the windowing system. ALT key
combinations cannot be mapped.
  menu	Using ALT in combination with a character that is a menu
shortcut key, will be handled by the windowing system. Other
keys can be mapped.
If the menu is disabled by excluding 'm' from 'guioptions', the ALT
key is never used for the menu.
This option is not used for <F10>; on Win32.

'winbar' 'wbr'

'winbar' 'wbr' string (default "")
global or local to window global-local

When non-empty, this option enables the window bar and determines its contents. The window bar is a bar that's shown at the top of every window with it enabled. The value of 'winbar' is evaluated like with 'statusline'.

When changing something that is used in 'winbar' that does not trigger it to be updated, use :redrawstatus.

Floating windows do not use the global value of 'winbar'. The window-local value of 'winbar' must be set for a floating window to have a window bar.

This option cannot be set in a modeline when 'modelineexpr' is off.

'winblend' 'winbl'

'winblend' 'winbl' number (default 0)
local to window

Enables pseudo-transparency for a floating window. Valid values are in the range of 0 for fully opaque window (disabled) to 100 for fully transparent background. Values between 0-30 are typically most useful.

UI-dependent. Works best with RGB colors. 'termguicolors'

'winborder'

'winborder' string (default "")
global

Defines the default border style of floating windows. The default value is empty, which is equivalent to "none". Valid values include:

- "bold": Bold line box.
- "double": Double-line box.
- "none": No border.
- "rounded": Like "single", but with rounded corners ("╭" etc.).
- "shadow": Drop shadow effect, by blending with the background.
- "single": Single-line box.
- "solid": Adds padding by a single whitespace cell.
- custom: comma-separated list of exactly 8 entries in clockwise

  order starting from topleft. Each entry may be a single char, a single space (filled with the background), or empty (no border on that side). Example:

```
vim.o.winborder = '+,-,+,|,+,-,+,|'
vim.o.winborder = ',,, ,,,, '  -- left/right padding only
```

'window' 'wi'

'window' 'wi' number (default screen height - 1)
global

Window height used for CTRL-F and CTRL-B when there is only one window and the value is smaller than 'lines' minus one. The screen will scroll 'window' minus two lines, with a minimum of one.
When 'window' is equal to 'lines' minus one CTRL-F and CTRL-B scroll in a much smarter way, taking care of wrapping lines.
When resizing the Vim window, and the value is smaller than 1 or more than or equal to 'lines' it will be set to 'lines' minus 1.
Note: Do not confuse this with the height of the Vim window, use 'lines' for that.

'winfixbuf' 'wfb' 'nowinfixbuf' 'nowfb'

'winfixbuf' 'wfb' boolean (default off)
local to window

If enabled, the window and the buffer it is displaying are paired. For example, attempting to change the buffer with :edit will fail. Other commands which change a window's buffer such as :cnext will also skip any window with 'winfixbuf' enabled. However if an Ex command has a "!" modifier, it can force switching buffers.

'winfixheight' 'wfh' 'nowinfixheight' 'nowfh'

'winfixheight' 'wfh' boolean (default off)
local to window local-noglobal

Keep the window height when windows are opened or closed and 'equalalways' is set. Also for CTRL-W =. Set by default for the preview-window and quickfix-window.
The height may be changed anyway when running out of room.

'winfixwidth' 'wfw' 'nowinfixwidth' 'nowfw'

'winfixwidth' 'wfw'	boolean (default off)
local to window local-noglobal

Keep the window width when windows are opened or closed and 'equalalways' is set. Also for CTRL-W =.
The width may be changed anyway when running out of room.

'winheight' 'wh' E591

'winheight' 'wh'	number (default 1)
global

Minimal number of lines for the current window. This is not a hard minimum, Vim will use fewer lines if there is not enough room. If the focus goes to a window that is smaller, its size is increased, at the cost of the height of other windows.
Set 'winheight' to a small number for normal editing.
Set it to 999 to make the current window fill most of the screen.
Other windows will be only 'winminheight' high. This has the drawback that ":all" will create only two windows. To avoid "vim -o 1 2 3 4" to create only two windows, set the option after startup is done, using the VimEnter event:

```
au VimEnter * set winheight=999
```

Minimum value is 1.
The height is not adjusted after one of the commands that change the height of the current window.
'winheight' applies to the current window. Use 'winminheight' to set the minimal height for other windows.

'winhighlight' 'winhl'

'winhighlight' 'winhl'	string (default "")
local to window

Window-local highlights. Comma-delimited list of highlight group-name pairs "{hl-from}:{hl-to},..." where each {hl-from} is a highlight-groups item to be overridden by {hl-to} group in the window.

Note: highlight namespaces take precedence over 'winhighlight'.
See nvim_win_set_hl_ns() and nvim_set_hl().

Highlights of vertical separators are determined by the window to the left of the separator. The 'tabline' highlight of a tabpage is decided by the last-focused window of the tabpage. Highlights of the popupmenu are determined by the current window. Highlights in the message area cannot be overridden.

Example: show a different color for non-current windows:

```
set winhighlight=Normal:MyNormal,NormalNC:MyNormalNC
```

'winminheight' 'wmh'

'winminheight' 'wmh'	number (default 1)
global

The minimal height of a window, when it's not the current window.
This is a hard minimum, windows will never become smaller.
When set to zero, windows may be "squashed" to zero lines (i.e. just a status bar) if necessary. They will return to at least one line when they become active (since the cursor has to have somewhere to go.)
Use 'winheight' to set the minimal height of the current window.
This option is only checked when making a window smaller. Don't use a large number, it will cause errors when opening more than a few windows. A value of 0 to 3 is reasonable.

'winminwidth' 'wmw'

'winminwidth' 'wmw'	number (default 1)
global

The minimal width of a window, when it's not the current window.
This is a hard minimum, windows will never become smaller.
When set to zero, windows may be "squashed" to zero columns (i.e. just a vertical separator) if necessary. They will return to at least one line when they become active (since the cursor has to have somewhere to go.)
Use 'winwidth' to set the minimal width of the current window.
This option is only checked when making a window smaller. Don't use a large number, it will cause errors when opening more than a few windows. A value of 0 to 12 is reasonable.

'winpinned' 'wp' 'nowinpinned' 'nowp'

'winpinned' 'wp'	boolean (default off)
local to window local-noglobal

If enabled, the window is pinned and will not be closed by :only and :fclose. Only commands specifically targeting the window can close it.

'winwidth' 'wiw' E592

'winwidth' 'wiw'	number (default 20)
global

Minimal number of columns for the current window. This is not a hard minimum, Vim will use fewer columns if there is not enough room. If the current window is smaller, its size is increased, at the cost of the width of other windows. Set it to 999 to make the current window always fill the screen. Set it to a small number for normal editing.
The width is not adjusted after one of the commands to change the width of the current window.

'winwidth' applies to the current window. Use 'winminwidth' to set
the minimal width for other windows.

'wrap' 'nowrap'

'wrap' boolean (default on)
local to window
This option changes how text is displayed. It doesn't change the text
in the buffer, see 'textwidth' for that.
When on, lines longer than the width of the window will wrap and
displaying continues on the next line. When off lines will not wrap
and only part of long lines will be displayed. When the cursor is
moved to a part that is not shown, the screen will scroll
horizontally.
The line will be broken in the middle of a word if necessary. See
'linebreak' to get the break at a word boundary.
To make scrolling horizontally a bit more useful, try this:

```
set sidescroll=5
set listchars+=precedes:<,extends:>
```

See 'sidescroll', 'listchars' and wrap-off.
This option can't be set from a modeline when the 'diff' option is
on.
If 'nowrap' was set from a modeline or in the sandbox, '>' is used
as the lcs-extends character regardless of the value of the 'list'
and 'listchars' options. This is to prevent malicious code outside
the viewport from going unnoticed. Use :setlocal nowrap manually
afterwards to disable this behavior.

'wrapmargin' 'wm'

'wrapmargin' 'wm' number (default 0)
local to buffer
Number of characters from the right window border where wrapping
starts. When typing text beyond this limit, an <EOL> will be inserted
and inserting continues on the next line.
Options that add a margin, such as 'number' and 'foldcolumn', cause
the text width to be further reduced.
When 'textwidth' is non-zero, this option is not used.
See also 'formatoptions' and ins-textwidth.

'wrapscan' 'ws' 'nowrapscan' 'nows' E384 E385

'wrapscan' 'ws' boolean (default on)
global
Searches wrap around the end of the file. Also applies to]s and
[s, searching for spelling mistakes.

'write' 'nowrite'

'write' boolean (default on)
global
Allows writing files. When not set, writing a file is not allowed.
Can be used for a view-only mode, where modifications to the text are
still allowed. Can be reset with the -m or -M command line
argument. Filtering text is still possible, even though this requires
writing a temporary file.

'writeany' 'wa' 'nowriteany' 'nowa'

'writeany' 'wa' boolean (default off)
global
Allows writing to any file with no need for "!" override.

'writebackup' 'wb' 'nowritebackup' 'nowb'

'writebackup' 'wb' boolean (default on)
global
Make a backup before overwriting a file. The backup is removed after
the file was successfully written, unless the 'backup' option is
also on.
WARNING: Switching this option off means that when Vim fails to write
your buffer correctly and then, for whatever reason, Vim exits, you
lose both the original file and what you were writing. Only reset
this option if your file system is almost full and it makes the write
fail (and make sure not to exit Vim until the write was successful).
See backup-table for another explanation.
When the 'backupskip' pattern matches, a backup is not made anyway.
Depending on 'backupcopy' the backup is a new file or the original
file renamed (and a new file is written).

'writedelay' 'wd'

'writedelay' 'wd' number (default 0)
global
Only takes effect together with 'redrawdebug'.
The number of milliseconds to wait after each line or each flush

Using Vim packages

A Vim "package" is a directory that contains plugins. Compared to normal plugins, a package can...
- be downloaded as an archive and unpacked in its own directory, so the files are not mixed with files of other plugins.
- be a git, mercurial, etc. repository, thus easy to update.
- contain multiple plugins that depend on each other.
- contain plugins that are automatically loaded on startup ("start" packages, located in "pack/*/start/*") and ones that are only loaded when needed with :packadd ("opt" packages, located in "pack/*/opt/*").

runtime-search-path

Nvim searches for :runtime files in:
- 1. all paths in 'runtimepath'
- 2. all "pack/*/start/*" dirs

Note that the "pack/*/start/*" paths are not explicitly included in 'runtimepath', so they will not be reported by ":set rtp" or "echo &rtp". Scripts can use nvim_list_runtime_paths() to list all used directories, and nvim_get_runtime_file() to query for specific files or sub-folders within the runtime path. Example:

```
" List all runtime dirs and packages with Lua paths.
:echo nvim_get_runtime_file("lua/", v:true)
```

Using a package and loading automatically

Let's assume your Nvim files are in "~/.local/share/nvim/site" and you want to add a package from a zip archive "/tmp/foopack.zip":

```
% mkdir -p ~/.local/share/nvim/site/pack/foo
% cd ~/.local/share/nvim/site/pack/foo
% unzip /tmp/foopack.zip
```

The directory name "foo" is arbitrary, you can pick anything you like.

You would now have these files under ~/.local/share/nvim/site:

```
pack/foo/README.txt
pack/foo/start/foobar/plugin/foo.vim
pack/foo/start/foobar/syntax/some.vim
pack/foo/opt/foodebug/plugin/debugger.vim
```

On startup after processing your config, Nvim scans all directories in 'packpath' for plugins in "pack/*/start/*", then loads the plugins.

To allow for calling into package functionality while parsing your vimrc, :colorscheme and autoload will both automatically search under 'packpath' as well in addition to 'runtimepath'. See the documentation for each for details.

In the example Nvim will find "pack/foo/start/foobar/plugin/foo.vim" and load it.

If the "foobar" plugin kicks in and sets the 'filetype' to "some", Nvim will find the syntax/some.vim file, because its directory is in the runtime search path.

Nvim will also load ftdetect files, if there are any.

Note that the files under "pack/foo/opt" are not loaded automatically, only the ones under "pack/foo/start". See pack-add below for how the "opt" directory is used.

Loading packages automatically will not happen if loading plugins is disabled, see load-plugins.

To load packages earlier, so that plugin/ files are sourced: :packloadall This also works when loading plugins is disabled. The automatic loading will only happen once.

If the package has an "after" directory, that directory is added to the end of 'runtimepath', so that anything there will be loaded later.

Using a single plugin and loading it automatically

If you don't have a package but a single plugin, you need to create the extra directory level:

```
% mkdir -p ~/.local/share/nvim/site/pack/foo/start/foobar
% cd ~/.local/share/nvim/site/pack/foo/start/foobar
% unzip /tmp/someplugin.zip
```

You would now have these files:

```
pack/foo/start/foobar/plugin/foo.vim
pack/foo/start/foobar/syntax/some.vim
```

From here it works like above.

Optional plugins

pack-add

To load an optional plugin from a pack use the `:packadd` command:

```
:packadd foodebug
```

This searches for "pack/*/opt/foodebug" in 'packpath' and will find ~/.local/share/nvim/site/pack/foo/opt/foodebug/plugin/debugger.vim and source it.

This could be done if some conditions are met. For example, depending on whether Nvim supports a feature or a dependency is missing.

You can also load an optional plugin at startup, by putting this command in your config:

```
:packadd! foodebug
```

The extra "!" is so that the plugin isn't loaded if Nvim was started with --noplugin.

It is perfectly normal for a package to only have files in the "opt" directory. You then need to load each plugin when you want to use it.

Where to put what

Since color schemes, loaded with `:colorscheme`, are found below "pack/*/start" and "pack/*/opt", you could put them anywhere. We recommend you put them below "pack/*/opt", for example "~/.config/nvim/pack/mycolors/opt/dark/colors/very_dark.vim".

Filetype plugins should go under "pack/*/start", so that they are always found. Unless you have more than one plugin for a file type and want to select which one to load with `:packadd`. E.g. depending on the compiler version:

```
if foo_compiler_version > 34
  packadd foo_new
else
  packadd foo_old
endif
```

The "after" directory is most likely not useful in a package. It's not disallowed though.

Creating Vim packages

package-create

This assumes you write one or more plugins that you distribute as a package.

If you have two unrelated plugins you would use two packages, so that Vim users can choose what they include or not. Or you can decide to use one package with optional plugins, and tell the user to add the preferred ones with `:packadd`.

Decide how you want to distribute the package. You can create an archive or you could use a repository. An archive can be used by more users, but is a bit harder to update to a new version. A repository can usually be kept up-to-date easily, but it requires a program like "git" to be available. You can do both, github can automatically create an archive for a release.

Your directory layout would be like this:

```
start/foobar/plugin/foo.vim          " always loaded, defines commands
start/foobar/plugin/bar.vim          " always loaded, defines commands
start/foobar/autoload/foo.vim        " loaded when foo command used
start/foobar/doc/foo.txt             " help for foo.vim
start/foobar/doc/tags                " help tags
opt/fooextra/plugin/extra.vim        " optional plugin, defines commands
opt/fooextra/autoload/extra.vim      " loaded when extra command used
opt/fooextra/doc/extra.txt           " help for extra.vim
opt/fooextra/doc/tags                " help tags
```

This allows for the user to do:

```
mkdir ~/.local/share/nvim/site/pack
cd ~/.local/share/nvim/site/pack
git clone https://github.com/you/foobar.git myfoobar
```

Here "myfoobar" is a name that the user can choose, the only condition is that it differs from other packages.

In your documentation you explain what the plugins do, and tell the user how to load the optional plugin:

```
:packadd! fooextra
```

You could add this packadd command in one of your plugins, to be executed when the optional plugin is needed.

Run the `:helptags` command to generate the doc/tags file. Including this generated file in the package means that the user can drop the package in the pack directory and the help command works right away. Don't forget to re-run the command after changing the plugin help:

```
:helptags path/start/foobar/doc
:helptags path/opt/fooextra/doc
```

Dependencies between plugins

packload-two-steps

Suppose you have two plugins that depend on the same functionality. You can put the common functionality in an autoload directory, so that it will be found automatically. Your package would have these files:

pack/foo/start/one/plugin/one.vim

```
call foolib#getit()
```

pack/foo/start/two/plugin/two.vim

```
call foolib#getit()
```

pack/foo/start/lib/autoload/foolib.vim

```
func foolib#getit()
```

This works, because start packages will be searched for autoload files, when sourcing the plugins.

Plugin manager

vim.pack

Install, update, and delete external plugins. **WARNING:** It is still considered experimental, yet should be stable enough for daily use.

Manages plugins only in a dedicated vim.pack-directory (see packages): `site/pack/core/opt` subdirectory of "data" standard-path. Subdirectory `site` of "data" standard path needs to be part of 'packpath'. It usually is, but might not be in cases like --clean or setting $XDG_DATA_HOME during startup. Plugin's subdirectory name matches plugin's name in specification. It is assumed that all plugins in the directory are managed exclusively by `vim.pack`.

Uses Git to manage plugins and requires present `git` executable. Target plugins should be Git repositories with versions as named tags following semver convention `v<major>.<minor>.<patch>` (with or without `v` prefix). Like `v1.2.0` or `1.2.0`, but not `1.2` or `v1`.

The latest state of all managed plugins is stored inside a vim.pack-lockfile located at 'packlockfile'. It is a JSON file that is used to persistently track data about plugins. For a more robust config treat lockfile like its part: put under version control, etc. In this case all plugins from the lockfile will be installed at once (in alphabetical order) and at lockfile's revision (instead of inferring from `version`). This is done on the very first `vim.pack` function call to ensure that lockfile is aligned with what is actually on the disk. Lockfile should not be edited by hand. Corrupted data for installed plugins is repaired (including after deleting whole file), but `version` fields will be missing for not yet added plugins.

vim.pack-examples

Basic install and management

- Add vim.pack.add() call(s) to 'init.lua':

```
vim.pack.add({
  -- Install "plugin1" and use default branch (usually `main` or `master`)
  'https://github.com/user/plugin1',

  -- Same as above, but using a table (allows setting other options)
  { src = 'https://github.com/user/plugin1' },

  -- Specify plugin's name (here the plugin will be called "plugin2"
  -- instead of "generic-name")
  { src = 'https://github.com/user/generic-name', name = 'plugin2' },

  -- Specify version to follow during install and update
  {
    src = 'https://github.com/user/plugin3',
    -- Version constraint, see |vim.version.range()|
    version = vim.version.range('1.0'),
  },
  {
    src = 'https://github.com/user/plugin4',
    -- Git branch, tag, or commit hash
    version = 'main',
  },
})

-- Plugin's code can be used directly after `add()`
plugin1 = require('plugin1')
```

- Restart Nvim (for example, with :restart). Plugins that were not yet installed will be available on disk after `add()` call. Their revision is taken from vim.pack-lockfile (if present) or inferred from the `version`.
- To update all plugins with new changes:
  - Execute :packupdate. This will download updates from source and show confirmation buffer in a separate tabpage.
  - Review changes. To confirm all updates execute :write. To discard updates execute :quit.
  - (Optionally) :restart to start using code from updated plugins.

Use shorter source

Create custom Lua helpers:

```
local gh = function(x) return 'https://github.com/' .. x end
local cb = function(x) return 'https://codeberg.org/' .. x end
vim.pack.add({ gh('user/plugin1'), cb('user/plugin2') })
```

Another approach is to utilize Git's `insteadOf` configuration:

- `git config --global url."https://github.com/".insteadOf "gh:"`
- `git config --global url."https://codeberg.org/".insteadOf "cb:"`
- In 'init.lua': `vim.pack.add({ 'gh:user/plugin1', 'cb:user/plugin2' })`. These sources will be used verbatim in vim.pack-lockfile, so reusing the config on different machine will require the same Git configuration.

Explore installed plugins

- `:packupdate ++offline`
- Navigate between plugins with `[[` and `]]`. List them with g0 (vim.lsp.buf.document_symbol()).

Switch plugin's version and/or source

- Update 'init.lua' for plugin to have desired `version` and/or `src`. Let's say, the switch is for plugin named 'plugin1'.
- :restart. The plugin's state on disk (revision and/or tracked source) is not yet changed. Only plugin's `version` in vim.pack-lockfile is updated.
- Execute `:packupdate plugin1`. The plugin's source is updated. If only switching version, also pass the `++offline` argument.
- Review changes and either confirm or discard them. If discarded, revert `version` change in 'init.lua' as well or you will be prompted again next time you run `:packupdate`.

Freeze plugin from being updated

- Update 'init.lua' for plugin to have `version` set to current revision. Get it from vim.pack-lockfile (plugin's field `rev`; looks like `abc12345`).
- :restart.

Unfreeze plugin to start receiving updates

- Update 'init.lua' for plugin to have `version` set to whichever version you want it to be updated.

- :restart.

Revert plugin after an update

- Revert the vim.pack-lockfile to the state before the update:
  - If Git tracked: `git checkout HEAD -- nvim-pack-lock.json`
  - If not tracked: examine log file ("nvim-pack.log" at "log" standard-path), locate the revisions before the latest update, and (carefully) adjust current lockfile to have those revisions.
- :restart.
- `:packupdate ++offline ++lockfile plugin`. Read and confirm.

Synchronize config across machines

- On main machine:
  - Add vim.pack-lockfile to VCS.
  - Push to the remote server.
- On secondary machine:
  - Pull from the server.
  - :restart. New plugins (not present locally, but present in the lockfile) are installed at proper revision. If some installation has failed but you know it should not (like due to bad Internet connection), revert vim.pack-lockfile and :restart again.
  - `:packupdate ++lockfile`. Read and confirm.
  - Manually delete outdated plugins (present locally, but were not present in the lockfile prior to restart) with `:packdel plugin`. They can be located by examining the VCS difference of the lockfile (`git diff -- nvim-pack-lock.json` for Git).

Remove plugins from disk

- Remove plugin specs from vim.pack.add() calls in 'init.lua' or they will be reinstalled later.
- :restart.
- Use :packdel with plugin names to remove. Use `:packdel ++all` to delete all inactive plugins.

Check for pending updates

- Run `vim.pack.get(nil, { offline = false })` and check the output for items with different `rev` and `rev_to` fields. To not download new updates from source, use plain `vim.pack.get()`.

Check for issues

- Run `:checkhealth vim.pack` and follow recommendations in case of detected issues.

Commands vim.pack-commands E5807

:packu :packupdate E5808 :packu[pdate][!] [++offline] [++lockfile] [name]

Interactively update the specified plugins. Skips confirmation when `!` is given. If no plugin names are provided, update all plugins.

When `++offline` is given, skip downloading new updates.

When `++lockfile` is given, use revisions from the lockfile.

:packd :packdel E5809 E5810 E5811 :packd[el][!] {name}

Remove the specified plugins. Can only remove inactive plugins unless `!` is given.

:packd[el][!] ++all

Remove all inactive plugins. When `!` is given, instead remove all plugins.

vim.pack-events

Performing actions via `vim.pack` functions can trigger these events:

- PackChangedPre - before trying to change plugin's state.
- PackChanged - after plugin's state has changed.

Events are triggered in bulk respecting order of plugins in which they are supplied. First all `PackChangedPre`, then perform all actions, and only after

- all `PackChanged` for successful actions. This provides more control for connected plugins, like by specifying dependencies before the plugin itself.

The event-data has these keys (type: `vim.event.packchanged.data`):

- `active` - whether plugin was added via vim.pack.add() to current session.
- `kind` - one of "install" (install on disk; before loading), "update" (update already installed plugin; might be not loaded), "delete" (delete from disk).
- `spec` - plugin's specification with defaults made explicit.
- `path` - full path to plugin's directory.

These events can be used to execute plugin hooks. For example:

```
local hooks = function(ev)
  -- Use available |event-data|
  local name, kind = ev.data.spec.name, ev.data.kind

  -- Run build script after plugin's code has changed
  if name == 'plug-1' and (kind == 'install' or kind == 'update') then
    -- Append `:wait()` if you need synchronous execution
    vim.system({ 'make' }, { cwd = ev.data.path })
  end

  -- If action relies on code from the plugin (like user command or
  -- Lua code), make sure to explicitly load it first
  if name == 'plug-2' and kind == 'update' then
    if not ev.data.active then
      vim.cmd.packadd('plug-2')
    end
    vim.cmd('PlugTwoUpdate')
    require('plug2').after_update()
  end
end

-- If hooks need to run on install, run this before `vim.pack.add()`
-- To act on install from lockfile, run before very first `vim.pack.add()`
vim.api.nvim_create_autocmd('PackChanged', { callback = hooks })
```

vim.pack-manifest

Plugins can come with a special top level `pkg.json` manifest file with extra information. If present, `vim.pack` uses it for improved user experience:

- Apply :source for scripts after triggering corresponding vim.pack-events. This allows plugins to define hooks that will be executed during plugin's lifetime. Sourcing is done with special context:
  - The current-directory is temporarily set to plugin's root (to make it easier to run vim.system() commands).
  - Plugin's path is temporarily ensured to be inside 'runtimepath' (so script can use require() with plugin's module, possibly with explicit package.loaded reset inside `"update"` scripts).
- Running :checkhealth for `vim.pack` will perform extra checks to ensure healthy plugin installation.

Full specification see at https://packspec.org/. See also vim.pack.Manifest. A simple example:

```
{
 "name": "best-plugin.nvim",
 "description": "The best plugin for Neovim",
 "engines": {
     "nvim": ">=0.13.0",
     "vim": ">=9.1.0"
 },
 "scripts": {
   "install": "scripts/install.lua",
   "preupdate": "scripts/preupdate.vim",
   "update": "scripts/update.vim",
   "preuninstall": "scripts/preuninstall.lua"
 }
}
```

vim.pack.Manifest

Fields:

- `{description}?` (`string`) Plugin description
- `{engines}?` (`table`) Supported engine versions. Values should be vim.version.range() compatible specs.
  - `{nvim}?` (`string`) Version range for Nvim.
  - `{vim}?` (`string`) Version range for Vim.
- `{name}?` (`string`) Plugin name
- `{scripts}?` (`table`) Script locations (relative to plugin's root) to :source after triggering corresponding vim.pack-events.
  - `{install}?` (`string`) Post install script.
  - `{preuninstall}?` (`string`) Pre delete script.
  - `{preupdate}?` (`string`) Pre update script.
  - `{update}?` (`string`) Post update script.

vim.pack.Spec

Fields:

- `{data}?` (any) Arbitrary data associated with a plugin.
- `{name}?` (`string`) Name of plugin. Will be used as directory name. Default: `src` repository name.
- `{src}` (`string`) URI from which to install and pull updates. Any format supported by `git clone` is allowed.
- `{version}?` (`string|vim.VersionRange`) Version to use for install and updates. Can be:
  - `nil` (no value, default) to use repository's default branch (usually `main` or `master`).
  - String to use specific branch, tag, or commit hash.
  - Output of vim.version.range() to install the greatest/last semver tag inside the version constraint.

add({specs}, {opts}) vim.pack.add()

Add plugin to current session

- For each specification check that plugin exists on disk in vim.pack-directory:
  - If exists, check if its `src` is the same as input. If not - delete immediately to clean install from the new source. Otherwise do nothing.
  - If doesn't exist, install it by downloading from `src` into `name` subdirectory (via partial blobless `git clone`) and update revision to match `version` (via `git checkout`). Plugin will not be on disk if any step resulted in an error.
- For each plugin execute :packadd (or customizable `load` function) making it reachable by Nvim.

Notes:
- Installation is done in parallel, but waits for all to finish before continuing next code execution.
- If plugin is already present on disk, there are no checks about its current revision. The specified `version` can be not the one actually present on disk. Execute vim.pack.update() to synchronize.
- Adding plugin second and more times during single session does nothing: only the data from the first adding is registered.

Parameters:
- `{specs}` (`(string|vim.pack.Spec)[]`) List of plugin specifications. String item is treated as `src`.
- `{opts}` (`table?`) A table with the following fields:
  - `{confirm}?` (`boolean`) Whether to ask user to confirm initial install. Default `true`.
  - `{load}?` (`boolean|fun(plug_data: {spec: vim.pack.Spec, path: string})`) Load `plugin/` files and `ftdetect/` scripts. If `false`, works like `:packadd!`. If function, called with plugin data and is fully responsible for loading plugin. Default `false` during init.lua sourcing and `true` afterwards.

del({names}, {opts}) vim.pack.del()
Remove plugins from disk

Parameters:
- `{names}` (`string[]`) List of plugin names to remove from disk. Must be managed by vim.pack, not necessarily already added to current session.
- `{opts}` (`table?`) A table with the following fields:
  - `{force}?` (`boolean`) Whether to allow deleting an active plugin. Default `false`.

get({names}, {opts}) vim.pack.get()
Gets vim.pack plugin info, optionally filtered by `names`.

Parameters:
- `{names}` (`string[]?`) List of plugin names. Default: all plugins managed by vim.pack.
- `{opts}` (`table?`) A table with the following fields:
  - `{info}?` (`boolean`) Whether to include extra plugin info. Default `true`.
  - `{offline}?` (`boolean`) Whether to skip downloading new updates. Requires `info=true`. Default: `true`.

Return:
(`table[]`) A list of objects with the following fields:
- `{active}` (`boolean`) Whether plugin was added via vim.pack.add() to current session.
- `{branches}?` (`string[]`) Available Git branches (first is default). Missing if `info=false`.
- `{manifest}?` (`vim.pack.Manifest`) Data from the vim.pack-manifest. Empty in case of reading error. Missing if `info=false`.
- `{path}` (`string`) Plugin's path on disk.
- `{rev}` (`string`) Current Git revision. Taken from vim.pack-lockfile if `info=false`.
- `{rev_to}?` (`string`) Git revision of a pending update. The same as used during vim.pack.update() and which points to a resolved `spec.version`. Missing if `info=false`.
- `{spec}` (`vim.pack.SpecResolved`) A vim.pack.Spec with resolved `name`.
- `{tags}?` (`string[]`) Available Git tags. Missing if `info=false`.

update({names}, {opts}) vim.pack.update()
Update plugins
- Download new changes from source.
- Infer update info (current/target revisions, changelog, etc.).
- If `force` is `false` (default), show confirmation buffer. If `force` is `true`, make updates right away.

Notes:
- Every actual update is logged in "nvim-pack.log" file inside "log" standard-path.
- It doesn't update source's default branch if it has changed (like from `master` to `main`). To have `version = nil` point to a new default branch, re-install the plugin (vim.pack.del() + vim.pack.add()).

Confirmation buffer

The goal of the confirmation buffer is to show update details for the user to read, confirm (execute :write) or deny (execute :quit) the update.

Pending changes starting with > will be applied while the ones starting with < will be reverted.

There are convenience buffer-local mappings:
-]] and [[to navigate through plugin sections.

Some features are provided via LSP:
- 'textDocument/documentLink' - compute links for plugin paths, sources, commits, and tags. Makes a best effort educated guess about a link structure. Use gx to open a link to an object at cursor.
- 'textDocument/documentSymbol' (g0 via lsp-defaults or vim.lsp.buf.document_symbol()) - show structure of the buffer.
- 'textDocument/hover' (K via lsp-defaults or vim.lsp.buf.hover()) - show more information at cursor. Like details of particular pending change or newer tag.
- 'textDocument/codeAction' (`gra` via lsp-defaults or vim.lsp.buf.code_action()) - show code actions relevant for "plugin at cursor". Like "delete" (after extra confirmation for active plugins), "update" or "skip updating" (if there are pending updates).

Parameters:
- `{names}` (`string[]?`) List of plugin names to update. Must be managed by vim.pack, not necessarily already added to current session. Default: names of all plugins managed by vim.pack.
- `{opts}` (`table?`) A table with the following fields:
  - `{force}?` (`boolean`) Whether to skip confirmation and make updates immediately. Default `false`.
  - `{offline}?` (`boolean`) Whether to skip downloading new updates. Default: `false`.
  - `{target}?` (`string`) How to compute a new plugin revision. One of:
    - "version" (default): use latest revision matching `version` from plugin specification.
    - "lockfile": use revision from the lockfile. For reverting or performing controlled update.

Patterns and search commands

The very basics can be found in section 03.9 of the user manual. A few more explanations are in chapter 27 usr_27.txt.

1. Search commands search-commands

/

/{pattern}[/]<CR> Search forward for the [count]'th occurrence of {pattern} exclusive.

/{pattern}/{offset}<CR> Search forward for the [count]'th occurrence of {pattern} and go {offset} lines up or down. linewise.

/<CR>

/<CR> Search forward for the [count]'th occurrence of the latest used pattern last-pattern with latest used {offset}.

//{offset}<CR> Search forward for the [count]'th occurrence of the latest used pattern last-pattern with new {offset}. If {offset} is empty no offset is used.

?

?{pattern}[?]<CR> Search backward for the [count]'th previous occurrence of {pattern} exclusive.

?{pattern}?{offset}<CR> Search backward for the [count]'th previous occurrence of {pattern} and go {offset} lines up or down linewise.

?<CR>

?<CR> Search backward for the [count]'th occurrence of the latest used pattern last-pattern with latest used {offset}.

??{offset}<CR> Search backward for the [count]'th occurrence of the latest used pattern last-pattern with new {offset}. If {offset} is empty no offset is used.

n

n Repeat the latest "/" or "?" [count] times.
If the cursor doesn't move the search is repeated with count + 1.
last-pattern

N

N Repeat the latest "/" or "?" [count] times in opposite direction. last-pattern

star E348 E349

* Search forward for the [count]'th occurrence of the word nearest to the cursor. The word used for the search is the first of:
1. the keyword under the cursor 'iskeyword'

2. the first keyword after the cursor, in the

current line
3. the non-blank word under the cursor

4. the first non-blank word after the cursor,

in the current line
Only whole keywords are searched for, like with the command "/\<keyword\>". exclusive
'ignorecase' is used, 'smartcase' is not.

v_star-default

{Visual}* In Visual mode, search forward for the current selection. default-mappings

#

Same as "*", but search backward. The pound sign (character 163) also works. If the "#" key works as backspace, try using "stty erase <BS>" before starting Vim (<BS> is CTRL-H or a real backspace).

v_#-default

{Visual}# In Visual mode, search backward for the current selection. default-mappings

gstar

g* Like "*", but don't put "\<" and "\>" around the word.
This makes the search also find matches that are not a
whole word.

g#

g# Like "#", but don't put "\<" and "\>" around the word.
This makes the search also find matches that are not a
whole word.

gd

gd Goto local Declaration. When the cursor is on a local
variable, this command will jump to its declaration.
This was made to work for C code, in other languages
it may not work well.
First Vim searches for the start of the current
function, just like "[[". If it is not found the
search stops in line 1. If it is found, Vim goes back
until a blank line is found. From this position Vim
searches for the keyword under the cursor, like with
"*", but lines that look like a comment are ignored
(see 'comments' option).
Note that this is not guaranteed to work, Vim does not
really check the syntax, it only searches for a match
with the keyword. If included files also need to be
searched use the commands listed in include-search.
After this command n searches forward for the next
match (not backward).

gD

gD Goto global Declaration. When the cursor is on a
global variable that is defined in the file, this
command will jump to its declaration. This works just
like "gd", except that the search for the keyword
always starts in line 1.

1gd

1gd Like "gd", but ignore matches inside a {} block that
ends before the cursor position.

1gD

1gD Like "gD", but ignore matches inside a {} block that
ends before the cursor position.

:noh[lsearch] :noh :nohlsearch
Stop the highlighting for the 'hlsearch' option. It
is automatically turned back on when using a search
command, or setting the 'hlsearch' option.
This command doesn't work in an autocommand, because
the highlighting state is saved and restored when
executing autocommands autocmd-searchpat.
Same thing for when invoking a user function.

While typing the search pattern the current match will be shown if the
'incsearch' option is on. Remember that you still have to finish the search
command with <CR> to actually position the cursor at the displayed match. Or
use <Esc> to abandon the search.

nohlsearch-auto

All matches for the last used search pattern will be highlighted if you set
the 'hlsearch' option. This can be suspended with the :nohlsearch command
or auto suspended with nohlsearch plugin. See nohlsearch-install.

When 'shortmess' does not include the "S" flag, Vim will automatically show an
index, on which the cursor is. This can look like this:

```
[1/5]           Cursor is on first of 5 matches.
[1/>99] Cursor is on first of more than 99 matches.
[>99/>99]       Cursor is after 99 match of more than 99 matches.
[?/??]  Unknown how many matches exists, generating the
              statistics was aborted because of search timeout.
```

Note: the count does not take offset into account.

When no match is found you get the error: E486 Pattern not found
Note that for the :global command, you get a normal message "Pattern not
found", for Vi compatibility.
For the :s command the "e" flag can be used to avoid the error message
:s_flags.

search-options

The following options affect how a search is performed in Vim:

'hlsearch'	highlight matches
'ignorecase'	ignore case when searching
'imsearch'	use IME when entering the search pattern
'incsearch'	show matches incrementally as the pattern is typed
'maxsearchcount'	maximum number for the search count shm-S
'shortmess'	suppress messages shm-s; show search count shm-S

'smartcase' override 'ignorecase' if pattern contains uppercase
'wrapscan' continue searching from the start of the file

search-offset {offset}

These commands search for the specified pattern. With "/" and "?" an additional offset may be given. There are two types of offsets: line offsets and character offsets.

The offset gives the cursor position relative to the found match:

```
[num]       [num] lines downwards, in column 1
+[num]      [num] lines downwards, in column 1
-[num]      [num] lines upwards, in column 1
e[+num]     [num] characters to the right of the end of the match
e[-num]     [num] characters to the left of the end of the match
s[+num]     [num] characters to the right of the start of the match
s[-num]     [num] characters to the left of the start of the match
b[+num]     [num] identical to s[+num] above (mnemonic: begin)
b[-num]     [num] identical to s[-num] above (mnemonic: begin)
;{pattern}  perform another search, see //;
```

If a '-' or '+' is given but [num] is omitted, a count of one will be used. When including an offset with 'e', the search becomes inclusive (the character the cursor lands on is included in operations).

Examples:

pattern	cursor position
/test/+1	one line below "test", in column 1
/test/e	on the last t of "test"
/test/s+2	on the 's' of "test"
/test/b-3	three characters before "test"

If one of these commands is used after an operator, the characters between the cursor position before and after the search is affected. However, if a line offset is given, the whole lines between the two cursor positions are affected.

An example of how to search for matches with a pattern and change the match with another word:

```
/foo<CR>        find "foo"
c//e<CR>        change until end of match
bar<Esc>        type replacement
//<CR>          go to start of next match
c//e<CR>        change until end of match
beep<Esc>       type another replacement
                etc.
```

//; E386

A very special offset is ';' followed by another search command. For example:

```
/test 1/;/test
/test.*/+1;?ing?
```

The first one first finds the next occurrence of "test 1", and then the first occurrence of "test" after that.

This is like executing two search commands after each other, except that:

- It can be used as a single motion command after an operator.
- The direction for a following "n" or "N" command comes from the first search command.
- When an error occurs the cursor is not moved at all.

last-pattern

The last used pattern and offset are remembered. They can be used to repeat the search, possibly in another direction or with another count. Note that two patterns are remembered: One for "normal" search commands and one for the substitute command ":s". Each time an empty pattern is given, the previously used pattern is used. However, if there is no previous search command, a previous substitute pattern is used, if possible.

The 'magic' option sticks with the last used pattern. If you change 'magic', this will not change how the last used pattern will be interpreted. The 'ignorecase' option does not do this. When 'ignorecase' is changed, it will result in the pattern to match other text.

All matches for the last used search pattern will be highlighted if you set the 'hlsearch' option.

To clear the last used search pattern:

```
:let @/ = ""
```

This will not set the pattern to an empty string, because that would match everywhere. The pattern is really cleared, like when starting Vim.

The search usually skips matches that don't move the cursor. Whether the next match is found at the next character or after the skipped match depends on the 'c' flag in 'cpoptions'. See cpo-c.

```
           with 'c' flag:   "/..." advances 1 to 3 characters
        without 'c' flag:   "/..." advances 1 character
```

The unpredictability with the 'c' flag is caused by starting the search in the first column, skipping matches until one is found past the cursor position.

When searching backwards, searching starts at the start of the line, using the 'c' flag in 'cpoptions' as described above. Then the last match before the cursor position is used.

In Vi the ":tag" command sets the last search pattern when the tag is searched for. In Vim this is not done, the previous search pattern is still remembered, unless the 't' flag is present in 'cpoptions'. The search pattern is always put in the search history.

If the 'wrapscan' option is on (which is the default), searches wrap around the end of the buffer. If 'wrapscan' is not set, the backward search stops at the beginning and the forward search stops at the end of the buffer. If 'wrapscan' is set and the pattern was not found the error message "pattern not found" is given, and the cursor will not be moved. If 'wrapscan' is not set the message becomes "search hit BOTTOM without match" when searching forward, or "search hit TOP without match" when searching backward. If wrapscan is set and the search wraps around the end of the file the message "search hit TOP, continuing at BOTTOM" or "search hit BOTTOM, continuing at TOP" is given when searching backwards or forwards respectively. This can be switched off by setting the 's' flag in the 'shortmess' option. The highlight method 'w' is used for this message (default: standout).

search-range

You can limit the search command "/" to a certain range of lines by including \%>l items. For example, to match the word "limit" below line 199 and above line 300:

```
/\%>199l\%<300llimit
```

Also see /\%>l.

Another way is to use the ":substitute" command with the 'c' flag. Example:

```
:.,300s/Pattern//gc
```

This command will search from the cursor position until line 300 for "Pattern". At the match, you will be asked to type a character. Type 'q' to stop at this match, type 'n' to find the next match.

The "*", "#", "g*" and "g#" commands look for a word near the cursor in this order, the first one that is found is used:

- The keyword currently under the cursor.
- The first keyword to the right of the cursor, in the same line.
- The WORD currently under the cursor.
- The first WORD to the right of the cursor, in the same line.

```
The keyword may only contain letters and characters in 'iskeyword'.
The WORD may contain any non-blanks (<Tab>s and/or <Space>s).
Note that if you type with ten fingers, the characters are easy to remember:
the "#" is under your left hand middle finger (search to the left and up) and
the "*" is under your right hand middle finger (search to the right and down).
(this depends on your keyboard layout though).
```

E956

In very rare cases a regular expression is used recursively. This can happen when executing a pattern takes a long time and when checking for messages on channels a callback is invoked that also uses a pattern or an autocommand is triggered. In most cases this should be fine, but if a pattern is in use when it's used again it fails. Usually this means there is something wrong with the pattern.

2. The definition of a pattern

search-pattern pattern [pattern]

regular-expression regexp Pattern
E383 E476

For starters, read chapter 27 of the user manual usr_27.txt.

/bar /\bar /pattern

1. A pattern is one or more branches, separated by "\|". It matches anything

   that matches one of the branches. Example: "foo\|beep" matches "foo" and matches "beep". If more than one branch matches, the first one is used.

```
  pattern ::=      branch
               or  branch \| branch
               or  branch \| branch \| branch
               etc.
```

/branch /\&

1. A branch is one or more concats, separated by "\&". It matches the last

    concat, but only if all the preceding concats also match at the same
    position. Examples:

```
"foobeep\&..." matches "foo" in "foobeep".
".*Peter\&.*Bob" matches in a line containing both "Peter" and "Bob"
```

```
    branch ::=       concat
                 or  concat \& concat
                 or  concat \& concat \& concat
                 etc.
```

/concat

1. A concat is one or more pieces, concatenated. It matches a match for the

    first piece, followed by a match for the second piece, etc. Example:
    "f[0-9]b", first matches "f", then a digit and then "b".

```
  concat  ::=      piece
               or  piece piece
               or  piece piece piece
               etc.
```

/piece

1. A piece is an atom, possibly followed by a multi, an indication of how many

    times the atom can be matched. Example: "a*" matches any sequence of "a"
    characters: "", "a", "aa", etc. See /multi.

```
  piece   ::=      atom
               or  atom  multi
```

/atom

1. An atom can be one of a long list of items. Many atoms match one character

    in the text. It is often an ordinary character or a character class.
    Parentheses can be used to make a pattern into an atom. The "\z(\)"
    construct is only for syntax highlighting.

```
  atom    ::=      ordinary-atom              /ordinary-atom
               or  \( pattern \)              /\(
               or  \%( pattern \)             /\%(
               or  \z( pattern \)             /\z(
```

/\%#= two-engines NFA

Vim includes two regexp engines:

1. An old, backtracking engine that supports everything.

2. A new, NFA engine that works much faster on some patterns, possibly slower

    on some patterns.

E1281

  Vim will automatically select the right engine for you. However, if you run
  into a problem or want to specifically select one engine or the other, you can
  prepend one of the following to the pattern:

```
      \%#=0   Force automatic selection.  Only has an effect when
              'regexpengine' has been set to a non-zero value.
      \%#=1   Force using the old engine.
      \%#=2   Force using the NFA engine.
```

You can also use the 'regexpengine' option to change the default.

E864 E868 E874 E875 E876 E877 E878

If selecting the NFA engine and it runs into something that is not implemented
the pattern will not match. This is only useful when debugging Vim.

3. Magic /magic

Some characters in the pattern, such as letters, are taken literally. They
match exactly the same character in the text. When preceded with a backslash
however, these characters may get a special meaning. For example, "a" matches
the letter "a", while "\a" matches any alphabetic character.

Other characters have a special meaning without a backslash. They need to be
preceded with a backslash to match literally. For example "." matches any
character while "\." matches a dot.

If a character is taken literally or not depends on the 'magic' option and the items in the pattern mentioned next. The 'magic' option should always be set. We mention the effect of 'nomagic' here for completeness, but we recommend against using that.

/\m /\M

Use of "\m" makes the pattern after it be interpreted as if 'magic' is set, ignoring the actual value of the 'magic' option.
Use of "\M" makes the pattern after it be interpreted as if 'nomagic' is used.

/\v /\V

Use of "\v" means that after it, all ASCII characters except '0'-'9', 'a'-'z', 'A'-'Z' and '_' have special meaning: "very magic"

Use of "\V" means that after it, only a backslash and the terminating character (usually / or ?) have special meaning: "very nomagic"

Examples:

after:	\v	\m 'magic'	\M 'nomagic'	\V	matches
	a	a	a	a	literal 'a'
	\a	\a	\a	\a	any alphabetic character
	.	.	\.	\.	any character
	\.	\.	.	.	literal dot
	$	$	$	\$	end-of-line
	*	*	*	*	any number of the previous atom
	~	~	\~	\~	latest substitute string
	()	\(\)	\(\)	\(\)	group as an atom
	|	|	|	|	nothing: separates alternatives
	\\	\\	\\	\\	literal backslash
	\{	{	{	{	literal curly brace

If you want to you can make a pattern immune to the 'magic' option being set or not by putting "\m" or "\M" at the start of the pattern.

4. Overview of pattern items pattern-overview

E865 E866 E867 E869

Overview of multi items. /multi E61 E62
More explanation and examples below, follow the links. E64 E871

multi 'magic'	'nomagic'		matches of the preceding atom	
/star	*	*	0 or more	as many as possible
/\+	\+	\+	1 or more	as many as possible
/\=	\=	\=	0 or 1	as many as possible
/\?	\?	\?	0 or 1	as many as possible
/\{	\{n,m}	\{n,m}	n to m	as many as possible
	\{n}	\{n}	n	exactly
	\{n,}	\{n,}	at least n	as many as possible
	\{,m}	\{,m}	0 to m	as many as possible
	\{}	\{}	0 or more	as many as possible (same as "*")
/\{-	\{-n,m}	\{-n,m}	n to m	as few as possible
	\{-n}	\{-n}	n	exactly
	\{-n,}	\{-n,}	at least n	as few as possible
	\{-,m}	\{-,m}	0 to m	as few as possible
	\{-}	\{-}	0 or more	as few as possible

E59

/\@>	\@>	\@>	1, like matching a whole pattern
/\@=	\@=	\@=	nothing, requires a match /zero-width
/\@!	\@!	\@!	nothing, requires NO match /zero-width
/\@<=	\@<=	\@<=	nothing, requires a match behind /zero-width
/\@<!	\@<!	\@<!	nothing, requires NO match behind /zero-width

Overview of ordinary atoms. /ordinary-atom

More explanation and examples below, follow the links.

ordinary atom magic	nomagic	matches	
/^	^	^	start-of-line (at start of pattern) /zero-width
/\^	\^	\^	literal '^'
/_^	_^	_^	start-of-line (used anywhere) /zero-width
/$	$	$	end-of-line (at end of pattern) /zero-width
/\$	\$	\$	literal '$'
/_$	_$	_$	end-of-line (used anywhere) /zero-width
/.	.	\.	any single character (not an end-of-line)
/_.	_.	_.	any single character or end-of-line
/\<	\<	\<	beginning of a word /zero-width
/\>	\>	\>	end of a word /zero-width
/\zs	\zs	\zs	anything, sets start of match
/\ze	\ze	\ze	anything, sets end of match
/\%^	\%^	\%^	beginning of file /zero-width E71
/\%$	\%$	\%$	end of file /zero-width
/\%V	\%V	\%V	inside Visual area /zero-width
/\%#	\%#	\%#	cursor position /zero-width
/\%'m	\%'m	\%'m	mark m position /zero-width
/\%l	\%23l	\%23l	in line 23 /zero-width
/\%c	\%23c	\%23c	in column 23 /zero-width
/\%v	\%23v	\%23v	in virtual column 23 /zero-width

Character classes: /character-classes

magic	nomagic	matches	
/\i	\i	\i	identifier character (see 'isident' option)
/\I	\I	\I	like "\i", but excluding digits
/\k	\k	\k	keyword character (see 'iskeyword' option)
/\K	\K	\K	like "\k", but excluding digits
/\f	\f	\f	file name character (see 'isfname' option)
/\F	\F	\F	like "\f", but excluding digits
/\p	\p	\p	printable character
/\P	\P	\P	like "\p", but excluding digits
/\s	\s	\s	whitespace character: <Space> and <Tab>
/\S	\S	\S	non-whitespace character; opposite of \s
/\d	\d	\d	digit: [0-9]
/\D	\D	\D	non-digit: [^0-9]
/\x	\x	\x	hex digit: [0-9A-Fa-f]
/\X	\X	\X	non-hex digit: [^0-9A-Fa-f]
/\o	\o	\o	octal digit: [0-7]
/\O	\O	\O	non-octal digit: [^0-7]
/\w	\w	\w	word character: [0-9A-Za-z_]
/\W	\W	\W	non-word character: [^0-9A-Za-z_]
/\h	\h	\h	head of word character: [A-Za-z_]
/\H	\H	\H	non-head of word character: [^A-Za-z_]
/\a	\a	\a	alphabetic character: [A-Za-z]
/\A	\A	\A	non-alphabetic character: [^A-Za-z]
/\l	\l	\l	lowercase character: [a-z]
/\L	\L	\L	non-lowercase character: [^a-z]
/\u	\u	\u	uppercase character: [A-Z]
/\U	\U	\U	non-uppercase character [^A-Z]
/_	_x	_x	where x is any of the characters above: character class with end-of-line included

(end of character classes)

magic	nomagic	matches		
/\e		\e	\e	<Esc>
/\t		\t	\t	<Tab>
/\r		\r	\r	<CR>
/\b		\b	\b	<BS>
/\n		\n	\n	end-of-line
/~		~	\~	last given substitute string
/\1		\1	\1	same string as matched by first \(\)
/\2		\2	\2	Like "\1", but uses second \(\) ...
/\9		\9	\9	Like "\1", but uses ninth \(\) E68
/\z1		\z1	\z1	only for syntax highlighting, see :syn-ext-match ...
/\z1		\z9	\z9	only for syntax highlighting, see :syn-ext-match

	x	x	a character with no special meaning matches itself
/[]	[]	\[]	any character specified inside the []
/\%[]	\%[]	\%[]	a sequence of optionally matched atoms
/\c	\c	\c	ignore case, do not use the 'ignorecase' option
/\C	\C	\C	match case, do not use the 'ignorecase' option
/\Z	\Z	\Z	ignore differences in Unicode "combining characters". Useful when searching voweled Hebrew or Arabic text.

magic	nomagic	matches		
/\m		\m	\m	'magic' on for the following chars in the pattern
/\M		\M	\M	'magic' off for the following chars in the pattern
/\v		\v	\v	the following chars in the pattern are "very magic"
/\V		\V	\V	the following chars in the pattern are "very nomagic" /\%#= \%#=1 \%#=1 select regexp engine /zero-width

/\%d	\%d	\%d	match specified decimal character (eg \%d123)
/\%x	\%x	\%x	match specified hex character (eg \%x2a)
/\%o	\%o	\%o	match specified octal character (eg \%o040)
/\%u	\%u	\%u	match specified multibyte character (eg \%u20ac)
/\%U	\%U	\%U	match specified large multibyte character (eg \%U12345678)
/\%C	\%C	\%C	match any composing characters

Example	matches
\<\I\i*	or\<\h\w*\<[a-zA-Z_][a-zA-Z0-9_]* An identifier (e.g., in a C program).

\(\.$|\. \)	A period followed by <EOL> or a space.
[.!?][])"']*\($|[ ]\)	A search pattern that finds the end of a sentence, with almost the same definition as the ")" command.
cat\Z	Both "cat" and "càt" ("a" followed by 0x0300) Does not match "càt" (character 0x00e0), even though it may look the same.

5. Multi items

pattern-multi-items

An atom can be followed by an indication of how many times the atom can be matched and in what way. This is called a multi. See /multi for an overview.

/star /\star

```
*       (use \* when 'magic' is not set)
        Matches 0 or more of the preceding atom, as many as possible.
        Example  'nomagic'      matches

 a*         a\*          "", "a", "aa", "aaa", etc.
 .*         \.\*         anything, also an empty string, no end-of-line
 \_.*       \_.\*        everything up to the end of the buffer
 \_.*END    \_.\*END     everything up to and including the last "END"
                         in the buffer
```

Exception: When "*" is used at the start of the pattern or just after "^" it matches the star character.

Be aware that repeating "_." can match a lot of text and take a long time. For example, _.*END matches all text from the current position to the last occurrence of "END" in the file. Since the "*" will match as many as possible, this first skips over all lines until the end of the file and then tries matching "END", backing up one character at a time.

/\+

```
\+      Matches 1 or more of the preceding atom, as many as possible.
Example         matches
        ^.\+$           any non-empty line
        \s\+            white space of at least one character
```

/\=

```
\=      Matches 0 or 1 of the preceding atom, as many as possible.
Example         matches
        foo\=           "fo" and "foo"
```

/\?

```
\?      Just like \=.  Cannot be used when searching backwards with the "?"
        command.
```

/\{ E60 E554 E870

```
\{n,m}  Matches n to m of the preceding atom, as many as possible
\{n}    Matches n of the preceding atom
\{n,}   Matches at least n of the preceding atom, as many as possible
\{,m}   Matches 0 to m of the preceding atom, as many as possible
\{}     Matches 0 or more of the preceding atom, as many as possible (like "*")
```

/\{-

```
\{-n,m} matches n to m of the preceding atom, as few as possible
\{-n}   matches n of the preceding atom
\{-n,}  matches at least n of the preceding atom, as few as possible
\{-,m}  matches 0 to m of the preceding atom, as few as possible
\{-}    matches 0 or more of the preceding atom, as few as possible
```

n and m are positive decimal numbers or zero

non-greedy

If a "-" appears immediately after the "{", then a shortest match first algorithm is used (see example below). In particular, "\{-}" is the same as "*" but uses the shortest match first algorithm. BUT: A match that starts earlier is preferred over a shorter match: "a\{-}b" matches "aaab" in "xaaab".

Example	matches
ab\{2,3}c	"abbc" or "abbbc"
a\{5}	"aaaaa"
ab\{2,}c	"abbc", "abbbc", "abbbbc", etc.
ab\{,3}c	"ac", "abc", "abbc" or "abbbc"
a[bc]\{3}d	"abbbd", "abbcd", "acbcd", "acccd", etc.
a\(bc\)\{1,2}d	"abcd" or "abcbcd"
a[bc]\{-}[cd]	"abc" in "abcd"
a[bc]*[cd]	"abcd" in "abcd"

The } may optionally be preceded with a backslash: \{n,m\}.

/\@=

```
\@=     Matches the preceding atom with zero width.
        Like "(?=pattern)" in Perl.
Example                 matches
        foo\(bar\)\@=           "foo" in "foobar"
        foo\(bar\)\@=foo        nothing
```

/zero-width

When using "\@=" (or "^", "$", "\<", "\>") no characters are included in the match. These items are only used to check if a match can be

made. This can be tricky, because a match with following items will be done in the same position. The last example above will not match "foobarfoo", because it tries match "foo" in the same position where "bar" matched.

Note that using "\&" works the same as using "\@=": "foo\&.." is the same as "\(foo\)\@=..". But using "\&" is easier, you don't need the parentheses.

/\@!

\@! Matches with zero width if the preceding atom does NOT match at the current position. /zero-width
Like "(?!pattern)" in Perl.

Example	**matches**
foo\(bar\)\@!	any "foo" not followed by "bar"
a.\{-}p\@!	"a", "ap", "app", "appp", etc. not immediately followed by a "p"
if \(\(then\)\@!.\)*$	"if " not followed by "then"

Using "\@!" is tricky, because there are many places where a pattern does not match. a.*p\@! will match from an "a" to the end of the line, because .* can match all characters in the line and the "p" doesn't match at the end of the line. "a.\{-}p\@!" will match any "a", "ap", "app", etc. that isn't followed by a "p", because the "." can match a "p" and "p\@!" doesn't match after that.

You can't use "\@!" to look for a non-match before the matching position: "\(foo\)\@!bar" will match "bar" in "foobar", because at the position where "bar" matches, "foo" does not match. To avoid matching "foobar" you could use "\(foo\)\@!...bar", but that doesn't match a bar at the start of a line. Use "\(foo\)\@<!bar".

Useful example: to find "foo" in a line that does not contain "bar":

```
/^\%(.*bar\)\@!.*\zsfoo
```

This pattern first checks that there is not a single position in the line where "bar" matches. If .*bar matches somewhere the \@! will reject the pattern. When there is no match any "foo" will be found. The "\zs" is to have the match start just before "foo".

/\@<=

\@<= Matches with zero width if the preceding atom matches just before what follows. /zero-width
Like "(?<=pattern)" in Perl, but Vim allows non-fixed-width patterns.

Example	**matches**
\(an_s\+\)\@<=file	"file" after "an" and white space or an end-of-line

For speed it's often much better to avoid this multi. Try using "\zs" instead /\zs. To match the same as the above example:
```
an\_s\+\zsfile
```
At least set a limit for the look-behind, see below.

"\@<=" and "\@<!" check for matches just before what follows. Theoretically these matches could start anywhere before this position. But to limit the time needed, only the line where what follows matches is searched, and one line before that (if there is one). This should be sufficient to match most things and not be too slow.

In the old regexp engine the part of the pattern after "\@<=" and "\@<!" are checked for a match first, thus things like "\1" don't work to reference \(\) inside the preceding atom. It does work the other way around:

Bad example	**matches**
\%#=1\1\@<=,\([a-z]\+\)	",abc" in "abc,abc"

However, the new regexp engine works differently, it is better to not rely on this behavior, do not use \@<= if it can be avoided:

Example	**matches**
\([a-z]\+\)\zs,\1	",abc" in "abc,abc"

\@123<=
Like "\@<=" but only look back 123 bytes. This avoids trying lots of matches that are known to fail and make executing the pattern very slow. Example, check if there is a "<" just before "span":
```
/<\@1<=span
```
This will try matching "<" only one byte before "span", which is the only place that works anyway.
After crossing a line boundary, the limit is relative to the end of the line. Thus the characters at the start of the line with the match are not counted (this is just to keep it simple).
The number zero is the same as no limit.

/\@<!

\@<! Matches with zero width if the preceding atom does NOT match just before what follows. Thus this matches if there is no position in the current or previous line where the atom matches such that it ends just before what follows. /zero-width
Like "(?<!pattern)" in Perl, but Vim allows non-fixed-width patterns.
The match with the preceding atom is made to end just before the match with what follows, thus an atom that ends in .* will work.
Warning: This can be slow (because many positions need to be checked

for a match). Use a limit if you can, see below.
Example matches

```
\(foo\)\@<!bar          any "bar" that's not in "foobar"
\(\/\/.*\)\@<!in        "in" which is not after "//"
```

\@123<!
Like "\@<!" but only look back 123 bytes. This avoids trying lots of matches that are known to fail and make executing the pattern very slow.

/\@>

\@> Matches the preceding atom like matching a whole pattern.
Like "(?>pattern)" in Perl.

Example	matches
\(a*\)\@>a	nothing (the "a*" takes all the "a"'s, there can't be another one following)

This matches the preceding atom as if it was a pattern by itself. If it doesn't match, there is no retry with shorter sub-matches or anything. Observe this difference: "a*b" and "a*ab" both match "aaab", but in the second case the "a*" matches only the first two "a"s. "\(a*\)\@>ab" will not match "aaab", because the "a*" matches the "aaa" (as many "a"s as possible), thus the "ab" can't match.

6. Ordinary atoms pattern-atoms

An ordinary atom can be:

/^

^ At beginning of pattern or after "\|", "\(", "\%(" or "\n": matches start-of-line; at other positions, matches literal '^'. /zero-width

Example	matches
^beep(	the start of the C function "beep" (probably).

/\^

\^ Matches literal '^'. Can be used at any position in the pattern, but not inside [].

/_^

_^ Matches start-of-line. /zero-width Can be used at any position in the pattern, but not inside [].

Example	matches
_s*_^foo	white space and blank lines and then "foo" at start-of-line

/$

$ At end of pattern or in front of "\|", "\)" or "\n" ('magic' on): matches end-of-line <EOL>; at other positions, matches literal '$'.
/zero-width

/\$

\$ Matches literal '$'. Can be used at any position in the pattern, but not inside [].

/_$

$ Matches end-of-line. /zero-width Can be used at any position in the pattern, but not inside []. Note that "a$b" never matches, since "b" cannot match an end-of-line. Use "a\nb" instead /\n.

Example	matches
foo_$_s*	"foo" at end-of-line and following white space and blank lines

. (with 'nomagic': \.) /. /\.
Matches any single character, but not an end-of-line.

/_.

_. Matches any single character or end-of-line.
Careful: _.* matches all text to the end of the buffer!

/\<

\< Matches the beginning of a word: The next char is the first char of a word. The 'iskeyword' option specifies what is a word character.
/zero-width

/\>

\> Matches the end of a word: The previous char is the last char of a word. The 'iskeyword' option specifies what is a word character.
/zero-width

/\zs

\zs Matches at any position, but not inside [], and sets the start of the match there: The next char is the first char of the whole match.
/zero-width
Example:

```
/^\s*\zsif
```

matches an "if" at the start of a line, ignoring white space.
Can be used multiple times, the last one encountered in a matching
branch is used. Example:

```
/\(.\{-}\zsFab\)\{3}
```

Finds the third occurrence of "Fab".
This cannot be followed by a multi. E888

/\ze

\ze Matches at any position, but not inside [], and sets the end of the match there: The previous char is the last char of the whole match. /zero-width
Can be used multiple times, the last one encountered in a matching branch is used.
Example: "end\ze\(if\|for\)" matches the "end" in "endif" and "endfor".
This cannot be followed by a multi. E888

/\%^ start-of-file

\%^ Matches start of the file. When matching with a string, matches the start of the string.
For example, to find the first "VIM" in a file:

```
/\%^\_.\{-}\zsVIM
```

/\%$ end-of-file

\%$ Matches end of the file. When matching with a string, matches the end of the string.
Note that this does NOT find the last "VIM" in a file:

```
/VIM\_.\{-}\%$
```

It will find the next VIM, because the part after it will always match. This one will find the last "VIM" in the file:

```
/VIM\ze\(\(VIM\)\@!\_.\)*\%$
```

This uses /\@! to ascertain that "VIM" does NOT match in any position after the first "VIM".
Searching from the end of the file backwards is easier!

/\%V

\%V Match inside the Visual area. When Visual mode has already been stopped match in the area that gv would reselect.
This is a /zero-width match. To make sure the whole pattern is inside the Visual area put it at the start and just before the end of the pattern, e.g.:

```
/\%Vfoo.*ba\%Vr
```

This also works if only "foo bar" was Visually selected. This:

```
/\%Vfoo.*bar\%V
```

would match "foo bar" if the Visual selection continues after the "r".
Only works for the current buffer.

/\%# cursor-position

\%# Matches with the cursor position. Only works when matching in a buffer displayed in a window.
WARNING: When the cursor is moved after the pattern was used, the result becomes invalid. Vim doesn't automatically update the matches.
This is especially relevant for syntax highlighting and 'hlsearch'.
In other words: When the cursor moves the display isn't updated for this change. An update is done for lines which are changed (the whole line is updated) or when using the CTRL-L command (the whole screen is updated). Example, to highlight the word under the cursor:

```
/\k*\%#\k*
```

When 'hlsearch' is set and you move the cursor around and make changes this will clearly show when the match is updated or not.

/\%'m /\%<'m /\%>'m

\%'m Matches with the position of mark m.
\%<'m Matches before the position of mark m.
\%>'m Matches after the position of mark m.
Example, to highlight the text from mark 's to 'e:

```
/.\%>'s.*\%<'e..
```

Note that two dots are required to include mark 'e in the match. That is because "\%<'e" matches at the character before the 'e mark, and since it's a /zero-width match it doesn't include that character.
WARNING: When the mark is moved after the pattern was used, the result becomes invalid. Vim doesn't automatically update the matches.
Similar to moving the cursor for "\%#" /\%#.

/\%l /\%>l /\%<l E951 E1204 E1273

\%23l   Matches in a specific line.
\%<23l Matches above a specific line (lower line number).
\%>23l Matches below a specific line (higher line number).
\%.l   Matches at the cursor line.
\%<.l   Matches above the cursor line.
\%>.l   Matches below the cursor line.
These six can be used to match specific lines in a buffer. The "23" can be any line number. The first line is 1.
WARNING: When inserting or deleting lines Vim does not automatically update the matches. This means Syntax highlighting quickly becomes wrong. Also when referring to the cursor position (".") and the cursor moves the display isn't updated for this change. An update is done when using the CTRL-L command (the whole screen is updated).
Example, to highlight the line where the cursor currently is:

```
:exe '/\%' .. line(".") .. 'l'
```

Alternatively use:

```
/\%.l
```

When 'hlsearch' is set and you move the cursor around and make changes this will clearly show when the match is updated or not.

/\%c /\%>c /\%<c

\%23c   Matches in a specific column.
\%<23c Matches before a specific column.
\%>23c Matches after a specific column.
\%.c   Matches at the cursor column.
\%<.c   Matches before the cursor column.
\%>.c   Matches after the cursor column.
These six can be used to match specific columns in a buffer or string. The "23" can be any column number. The first column is 1. Actually, the column is the byte number (thus it's not exactly right for multibyte characters).
WARNING: When inserting or deleting text Vim does not automatically update the matches. This means Syntax highlighting quickly becomes wrong. Also when referring to the cursor position (".") and the cursor moves the display isn't updated for this change. An update is done when using the CTRL-L command (the whole screen is updated).
Example, to highlight the column where the cursor currently is:

```
:exe '/\%' .. col(".") .. 'c'
```

Alternatively use:

```
/\%.c
```

When 'hlsearch' is set and you move the cursor around and make changes this will clearly show when the match is updated or not.
Example for matching a single byte in column 44:

```
/\%>43c.\%<46c
```

Note that "\%<46c" matches in column 45 when the "." matches a byte in column 44.

/\%v /\%>v /\%<v

\%23v   Matches in a specific virtual column.
\%<23v Matches before a specific virtual column.
\%>23v Matches after a specific virtual column.
\%.v   Matches at the current virtual column.
\%<.v   Matches before the current virtual column.
\%>.v   Matches after the current virtual column.
These six can be used to match specific virtual columns in a buffer or string. When not matching with a buffer in a window, the option values of the current window are used (e.g., 'tabstop').
The "23" can be any column number. The first column is 1.
Note that some virtual column positions will never match, because they are halfway through a tab or other character that occupies more than one screen character.
WARNING: When inserting or deleting text Vim does not automatically update highlighted matches. This means Syntax highlighting quickly becomes wrong. Also when referring to the cursor position (".") and the cursor moves the display isn't updated for this change. An update is done when using the CTRL-L command (the whole screen is updated).
Example, to highlight all the characters after virtual column 72:

```
/\%>72v.*
```

When 'hlsearch' is set and you move the cursor around and make changes this will clearly show when the match is updated or not.
To match the text up to column 17:

```
/^.*\%17v
```

To match all characters after the current virtual column (where the cursor is):

```
/\%>.v.*
```

Column 17 is not included, because this is a /zero-width match. To include the column use:

```
/^.*\%17v.
```

This command does the same thing, but also matches when there is no character in column 17:

```
/^.*\%<18v.
```

Note that without the "^" to anchor the match in the first column, this will also highlight column 17:

```
/.*\%17v
```

Column 17 is highlighted by 'hlsearch' because there is another match where .* matches zero characters.

Character classes:

\i	identifier character (see 'isident' option)	/\i
\I	like "\i", but excluding digits	/\I
\k	keyword character (see 'iskeyword' option)	/\k
\K	like "\k", but excluding digits	/\K
\f	file name character (see 'isfname' option)	/\f
\F	like "\f", but excluding digits	/\F
\p	printable character	/\p
\P	like "\p", but excluding digits	/\P

NOTE: the above also work for multibyte characters. The ones below only match ASCII characters, as indicated by the range.

whitespace white-space

\s	whitespace character: <Space> and <Tab>		/\s
\S	non-whitespace character; opposite of \s		/\S
\d	digit:	[0-9]	/\d
\D	non-digit:	[^0-9]	/\D
\x	hex digit:	[0-9A-Fa-f]	/\x
\X	non-hex digit:	[^0-9A-Fa-f]	/\X
\o	octal digit:	[0-7]	/\o
\O	non-octal digit:	[^0-7]	/\O
\w	word character:	[0-9A-Za-z_]	/\w
\W	non-word character:	[^0-9A-Za-z_]	/\W
\h	head of word character:	[A-Za-z_]	/\h
\H	non-head of word character:	[^A-Za-z_]	/\H
\a	alphabetic character:	[A-Za-z]	/\a
\A	non-alphabetic character:	[^A-Za-z]	/\A
\l	lowercase character:	[a-z]	/\l
\L	non-lowercase character:	[^a-z]	/\L
\u	uppercase character:	[A-Z]	/\u
\U	non-uppercase character:	[^A-Z]	/\U

NOTE: Using the atom is faster than the [] form.

NOTE: 'ignorecase', "\c" and "\C" are not used by character classes.

/_ E63 /_i /_I /_k /_K /_f /_F
/_p /_P /_s /_S /_d /_D /_x /_X
/_o /_O /_w /_W /_h /_H /_a /_A
/_l /_L /_u /_U

_x Where "x" is any of the characters above: The character class with end-of-line added

(end of character classes)

\e	matches <Esc>	/\e
\t	matches <Tab>	/\t
\r	matches <CR>	/\r
\b	matches <BS>	/\b
\n	matches an end-of-line	/\n

When matching in a string instead of buffer text a literal newline character is matched.

~	matches the last given substitute string	/~ /\~

\(\) A pattern enclosed by escaped parentheses. /\(/\(\) /\)
E.g., "\(^a\)" matches 'a' at the start of a line.
There can only be nine of these. You can use "\%(" to add more, but not counting it as a sub-expression.
E51 E54 E55 E872 E873

\1	Matches the same string that was matched by	/\1 E65
	the first sub-expression in \(and \).
	Example: "\([a-z]\).\1" matches "ata", "ehe", "tot", etc.
\2	Like "\1", but uses second sub-expression,	/\2
...	/\3

\9	Like "\1", but uses ninth sub-expression.	/\9
	Note: The numbering of groups is done based on which "\(" comes first
	in the pattern (going left to right), NOT based on what is matched
	first.

\%(\)	A pattern enclosed by escaped parentheses.	/\%(\) /\%(E53
	Just like \(\), but without counting it as a sub-expression. This
	allows using more groups and it's a little bit faster.

x	A single character, with no special meaning, matches itself
	/\ /\\

\x	A backslash followed by a single character, with no special meaning,
	is reserved for future expansions

[]	(with 'nomagic': \[])	/[] /\[] /_[] /collection E76
_[]
	A collection. This is a sequence of characters enclosed in square
	brackets. It matches any single character in the collection.

Example	**matches**
	[xyz]	any 'x', 'y' or 'z'
	[a-zA-Z]$	any alphabetic character at the end of a line
	\c[a-z]$	same
	[A-яЁё]	Russian alphabet (with utf-8 and cp1251)

/[\n]

	With "_" prepended the collection also includes the end-of-line.
	The same can be done by including "\n" in the collection. The
	end-of-line is also matched when the collection starts with "^"! Thus
	"_[^ab]" matches the end-of-line and any character but "a" and "b".
	This makes it Vi compatible: Without the "_" or "\n" the collection
	does not match an end-of-line.

E769

	When the ']' is not there Vim will not give an error message but
	assume no collection is used. Useful to search for '['. However, you
	do get E769 for internal searching. And be aware that in a
	:substitute command the whole command becomes the pattern. E.g.
	":s/[/x/" searches for "[/x" and replaces it with nothing. It does
	not search for "[" and replaces it with "x"!

E944 E945

	If the sequence begins with "^", it matches any single character NOT
	in the collection: "[^xyz]" matches anything but 'x', 'y' and 'z'.

- If two characters in the sequence are separated by '-', this is

	shorthand for the full list of ASCII characters between them. E.g.,
	"[0-9]" matches any decimal digit. If the starting character
	exceeds the ending character, e.g. [c-a], E944 occurs. Non-ASCII
	characters can be used, but the character values must not be more
	than 256 apart in the old regexp engine. For example, searching by
	[\u3000-\u4000] after setting re=1 emits a E945 error. Prepending
	\%#=2 will fix it.

- A character class expression is evaluated to the set of characters

	belonging to that character class. The following character classes
	are supported:

	Name	Func	Contents ~
[:alnum:]	[:alnum:]	isalnum	ASCII letters and digits
[:alpha:]	[:alpha:]	isalpha	ASCII letters
[:blank:]	[:blank:]		space and tab
[:cntrl:]	[:cntrl:]	iscntrl	ASCII control characters
[:digit:]	[:digit:]		decimal digits '0' to '9'
[:graph:]	[:graph:]	isgraph	ASCII printable characters excluding space
[:lower:]	[:lower:]	(1)	lowercase letters (all letters when 'ignorecase' is used and the old engine is in use two-engines)
[:print:]	[:print:]	(2)	printable characters including space
[:punct:]	[:punct:]	ispunct	ASCII punctuation characters
[:space:]	[:space:]		whitespace characters: space, tab, CR, NL, vertical tab, form feed
[:upper:]	[:upper:]	(3)	uppercase letters (all letters when 'ignorecase' is used and the old engine is in use two-engines)
[:xdigit:]	[:xdigit:]		hexadecimal digits: 0-9, a-f, A-F
[:return:]	[:return:]		the <CR> character
[:tab:]	[:tab:]		the <Tab> character
[:escape:]	[:escape:]		the <Esc> character
[:backspace:]	[:backspace:]		the <BS> character
[:ident:]	[:ident:]		identifier character (same as "\i")
[:keyword:]	[:keyword:]		keyword character (same as "\k")
[:fname:]	[:fname:]		file name character (same as "\f")

	The square brackets in character class expressions are additional to

the square brackets delimiting a collection. For example, the following is a plausible pattern for a UNIX filename: "[-./[:alnum:]_~]\+". That is, a list of at least one character, each of which is either '-', '.', '/', alphabetic, numeric, '_' or '~'.
These items only work for 8-bit characters, except [:lower:] and [:upper:] also work for multibyte characters when using the new regexp engine. See two-engines. In the future these items may work for multibyte characters. For now, to get all "alpha" characters you can use: [[:lower:][:upper:]].

The "Func" column shows what library function is used. The implementation depends on the system. Otherwise:
(1) Uses islower() for ASCII and Vim builtin rules for other characters.
(2) Uses Vim builtin rules
(3) As with (1) but using isupper()

/[[= [==]

- An equivalence class. This means that characters are matched that

have almost the same meaning, e.g., when ignoring accents. This only works for Unicode, latin1 and latin9. The form is:
[=a=]

/[[. [..]

- A collation element. This currently simply accepts a single

character in the form:
[.a.]

/\]

- To include a literal ']', '^', '-' or '\' in the collection, put a

backslash before it: "[xyz\]]", "[\^xyz]", "[xy\-z]" and "[xyz\\]". (**Note:** POSIX does not support the use of a backslash this way). For ']' you can also make it the first character (following a possible "^"): "[]xyz]" or "[^]xyz]".
For '-' you can also make it the first or last character: "[-xyz]", "[^-xyz]" or "[xyz-]". For '\' you can also let it be followed by any character that's not in "^]-\bdertnoUux". "[\xyz]" matches '\', 'x', 'y' and 'z'. It's better to use "\\" though, future expansions may use other characters after '\'.
- Omitting the trailing] is not considered an error. "[]" works like

"[]]", it matches the ']' character.
- The following translations are accepted when the 'l' flag is not

included in 'cpoptions':
\e <Esc>
\t <Tab>
\r <CR> (NOT end-of-line!)
\b <BS>
\n line break, see above /[\n]
\d123 decimal number of character
\o40 octal number of character up to 0o377
\x20 hexadecimal number of character up to 0xff
\u20AC hex. number of multibyte character up to 0xffff
\U1234 hex. number of multibyte character up to 8 characters
0xffffffff E1541
NOTE: The other backslash codes mentioned above do not work inside []!
- Matching with a collection can be slow, because each character in

the text has to be compared with each character in the collection. Use one of the other atoms above when possible. Example: "\d" is much faster than "[0-9]" and matches the same characters. However, the new NFA regexp engine deals with this better than the old one.

/\%[] E69 E70 E369

\%[] A sequence of optionally matched atoms. This always matches.
It matches as much of the list of atoms it contains as possible. Thus it stops at the first atom that doesn't match. For example:

```
/r\%[ead]
```

matches "r", "re", "rea" or "read". The longest that matches is used.
To match the Ex command "function", where "fu" is required and "nction" is optional, this would work:

```
/\<fu\%[nction]\>
```

The end-of-word atom "\>" is used to avoid matching "fu" in "full".
It gets more complicated when the atoms are not ordinary characters.
You don't often have to use it, but it is possible. Example:

```
/\<r\%[[eo]ad]\>
```

Matches the words "r", "re", "ro", "rea", "roa", "read" and "road".
There can be no \(\), \%(\) or \z(\) items inside the [] and \%[] does
not nest.
To include a "[" use "[[]" and for "]" use []]", e.g.,:

```
/index\%[[[]0[]]]
```

matches "index" "index[", "index[0" and "index[0]".

/\%d /\%x /\%o /\%u /\%U E678

\%d123 Matches the character specified with a decimal number. Must be
followed by a non-digit.
\%o40 Matches the character specified with an octal number up to 0o377.
Numbers below 0o40 must be followed by a non-octal digit or a
non-digit.
\%x2a Matches the character specified with up to two hexadecimal characters.
\%u20AC Matches the character specified with up to four hexadecimal
characters.
\%U1234abcd Matches the character specified with up to eight hexadecimal
characters, up to 0x7fffffff (the maximum allowed value is INT_MAX
E1541, but the maximum valid Unicode codepoint is U+10FFFF).

7. Ignoring case in a pattern /ignorecase

If the 'ignorecase' option is on, the case of normal letters is ignored.
'smartcase' can be set to ignore case when the pattern contains lowercase
letters only.

/\c /\C

When "\c" appears anywhere in the pattern, the whole pattern is handled like
'ignorecase' is on. The actual value of 'ignorecase' and 'smartcase' is
ignored. "\C" does the opposite: Force matching case for the whole pattern.
Note that 'ignorecase', "\c" and "\C" are not used for the character classes.

Examples:

pattern	'ignorecase'	'smartcase'	matches
foo	off	-	foo
foo	on	-	foo Foo FOO
Foo	on	off	foo Foo FOO
Foo	on	on	Foo
\cfoo	-	-	foo Foo FOO
foo\C	-	-	foo

Technical detail: NL-used-for-Nul

<Nul> characters in the file are stored as <NL> in memory. In the display
they are shown as "^@". The translation is done when reading and writing
files. To match a <Nul> with a search pattern you can just enter CTRL-@ or
"CTRL-V 000". This is probably just what you expect. Internally the
character is replaced with a <NL> in the search pattern. What is unusual is
that typing CTRL-V CTRL-J also inserts a <NL>, thus also searches for a <Nul>
in the file.

CR-used-for-NL

When 'fileformat' is "mac", <NL> characters in the file are stored as <CR>
characters internally. In the text they are shown as "^J". Otherwise this
works similar to the usage of <NL> for a <Nul>.

When working with expression evaluation, a <NL> character in the pattern
matches a <NL> in the string. The use of "\n" (backslash n) to match a <NL>
doesn't work there, it only works to match text in the buffer.

pattern-multi-byte pattern-multibyte

Patterns will also work with multibyte characters, mostly as you would
expect. But invalid bytes may cause trouble, a pattern with an invalid byte
will probably never match.

8. Composing characters patterns-composing

/\Z

When "\Z" appears anywhere in the pattern, all composing characters are
ignored. Thus only the base characters need to match, the composing
characters may be different and the number of composing characters may differ.
Exception: If the pattern starts with one or more composing characters, these
must match.

/\%C

Use "\%C" to skip any composing characters. For example, the pattern "a" does
not match in "càt" (where the a has the composing character 0x0300), but
"a\%C" does. Note that this does not match "cát" (where the á is character
0xe1, it does not have a compositing character). It does match "cat" (where
the a is just an a).

When a composing character appears at the start of the pattern or after an
item that doesn't include the composing character, a match is found at any
character that includes this composing character.

When using a dot and a composing character, this works the same as the
composing character by itself, except that it doesn't matter what comes before
this.

The order of composing characters does not matter. Also, the text may have more composing characters than the pattern, it still matches. But all composing characters in the pattern must be found in the text.

Suppose B is a base character and x and y are composing characters:

pattern	text	match
Bxy	Bxy	yes (perfect match)
Bxy	Byx	yes (order ignored)
Bxy	By	no (x missing)
Bxy	Bx	no (y missing)
Bx	Bx	yes (perfect match)
Bx	By	no (x missing)
Bx	Bxy	yes (extra y ignored)
Bx	Byx	yes (extra y ignored)

9. Compare with Perl patterns

perl-patterns

Vim's regexes are most similar to Perl's, in terms of what you can do. The difference between them is mostly just notation; here's a summary of where they differ:

Capability	in Vimspeak	in Perlspeak
force case insensitivity	\c	(?i)
force case sensitivity	\C	(?-i)
backref-less grouping	\%(atom\)	(?:atom)
conservative quantifiers	\{-n,m}	*?, +?, ??, {}?
0-width match	atom\@=	(?=atom)
0-width non-match	atom\@!	(?!atom)
0-width preceding match	atom\@<=	(?<=atom)
0-width preceding non-match	atom\@<!	(?<!atom)
match without retry	atom\@>	(?>atom)

Vim and Perl handle newline characters inside a string a bit differently:

In Perl, ^ and $ only match at the very beginning and end of the text, by default, but you can set the 'm' flag, which lets them match at embedded newlines as well. You can also set the 's' flag, which causes a . to match newlines as well. (Both these flags can be changed inside a pattern using the same syntax used for the i flag above, BTW.)

On the other hand, Vim's ^ and $ always match at embedded newlines, and you get two separate atoms, \%^ and \%$, which only match at the very start and end of the text, respectively. Vim solves the second problem by giving you the _ "modifier": put it in front of a . or a character class, and they will match newlines as well.

Finally, these constructs are unique to Perl:

- execution of arbitrary code in the regex: (?{perl code})
- conditional expressions: (?(condition)true-expr|false-expr)

...and these are unique to Vim:

- changing the magic-ness of a pattern: \v \V \m \M

  (very useful for avoiding backslashitis)
- sequence of optionally matching atoms: \%[atoms]
- \& (which is to \| what "and" is to "or"; it forces several branches

  to match at one spot)
- matching lines/columns by number: \%5l \%5c \%5v
- setting the start and end of the match: \zs \ze

10. Highlighting matches

match-highlight

syntax-vs-match

Note that the match highlight mechanism is independent of syntax-highlighting, which is (usually) a buffer-local highlighting, while matching is window-local, both methods can be freely mixed. Match highlighting functions give you a bit more flexibility in when and how to apply, but are typically only used for temporary highlighting, without strict rules. Both methods can be used to conceal text.

Thus the matching functions like matchadd() won't consider syntax rules and functions like synconcealed() and the other way around.

:mat :match

:mat[ch] {group} /{pattern}/
Define a pattern to highlight in the current window. It will be highlighted with {group}. Example:

```
:highlight MyGroup ctermbg=green guibg=green
:match MyGroup /TODO/
```

Instead of // any character can be used to mark the start and end of the {pattern}. Watch out for using special characters, such as '"' and '|'.

{group} must exist at the moment this command is executed.

The {group} highlighting still applies when a character is to be highlighted for 'hlsearch', as the highlighting for matches is given higher priority than that of 'hlsearch'. Syntax highlighting (see 'syntax') is also overruled by matches.

Note that highlighting the last used search pattern with 'hlsearch' is used in all windows, while the pattern defined with ":match" only exists in the current window. It is kept when switching to another buffer.

'ignorecase' does not apply, use /\c in the pattern to ignore case. Otherwise case is not ignored.

'redrawtime' defines the maximum time searched for pattern matches.

When matching end-of-line and Vim redraws only part of the display you may get unexpected results. That is because Vim looks for a match in the line where redrawing starts.

Also see matcharg() and getmatches(). The former returns the highlight group and pattern of a previous :match command. The latter returns a list with highlight groups and patterns defined by both matchadd() and :match.

Highlighting matches using :match are limited to three matches (aside from :match, :2match and :3match are available). matchadd() does not have this limitation and in addition makes it possible to prioritize matches.

Another example, which highlights all characters in virtual column 72 and more:

```
:highlight rightMargin term=bold ctermfg=blue guifg=blue
:match rightMargin /.\%>72v/
```

To highlight all character that are in virtual column 7:

```
:highlight col8 ctermbg=grey guibg=grey
:match col8 /\%<8v.\%>7v/
```

Note the use of two items to also match a character that occupies more than one virtual column, such as a TAB.

:mat[ch]
:mat[ch] none
Clear a previously defined match pattern.

:2mat[ch] {group} /{pattern}/ :2match

:2mat[ch]
:2mat[ch] none
:3mat[ch] {group} /{pattern}/ :3match

:3mat[ch]
:3mat[ch] none
Just like :match above, but set a separate match. Thus there can be three matches active at the same time. The match with the lowest number has priority if several match at the same position. It uses the match id 3.
The ":3match" command is used by (older Vims) matchparen plugin. You are suggested to use ":match" for manual matching and ":2match" for another plugin or even better make use of the more flexible matchadd() (and similar) functions instead.

11. Fuzzy matching fuzzy-matching

Fuzzy matching scores how well a string matches a pattern when the pattern characters appear in order but not necessarily contiguously.

Example:

```
Pattern:        "vim"
Candidates:     "vim"        -> perfect
            "vimeo"          -> good (v i m)
            "voice mail" -> weaker (v _ i _ _ _ m)
            "vintage"    -> no match (no "m")
```

If the search string has multiple words, each word is matched separately and may appear in any order in the candidate. For example "get pat" matches

"GetPattern", "PatternGet", "getPattern", "patGetter", "getSomePattern", "MatchpatternGet", etc.

The 'ignorecase' and 'smartcase' options do not apply, case is ignored if the pattern is all lower case.

Vim's implementation is based on the algorithm from the fzy project:
https://github.com/jhawthorn/fzy

It uses dynamic programming to compute an optimal score for a given pattern and candidate.

The algorithm works in two stages:

1. Forward pass

   Scan the candidate left to right, tracking the best score for each pattern position. Matches score higher when they occur at the start of the candidate, the start of a word (space, underscore, dash, camelCase), or directly after the previous match.
2. Backward pass

   Start from the best-scoring end position and step back to find match positions, ensuring the alignment is optimal.

Vim extends the original algorithm to support multibyte codepoints, allowing correct matching for UTF-8 and other encodings.

Time complexity is O(pattern * candidate). Memory usage is proportional to the same.

The matchfuzzy() and matchfuzzypos() functions perform fuzzy searching in a List of strings. matchfuzzy() returns the matching strings, while matchfuzzypos() returns the matches along with their positions and scores.

The "f" flag of :vimgrep enables fuzzy matching.

To enable fuzzy matching for ins-completion, add "fuzzy" to the 'completeopt' option. For cmdline-completion, add "fuzzy" to the 'wildoptions' option.

Editing compressed files with Vim bzip2 compress

1. Autocommands gzip-autocmd

The functionality mentioned here is a standard-plugin.
This plugin is only available if 'compatible' is not set.
You can avoid loading this plugin by setting the "loaded_gzip" variable:

```
:let loaded_gzip = 1
```

g:gzip_exec

For security reasons, one may prevent that Vim runs executables automatically
when opening a buffer. This option (default: "1") can be used to prevent
executing the executables command when set to "0":

```
:let g:gzip_exec = 0
```

1. Autocommands gzip-autocmd

The plugin installs autocommands to intercept reading and writing of files
with these extensions:

extension	compression
*.bz2	bzip2
*.bz3	bzip3
*.gz	gzip
*.lz	lzip
*.lz4	lz4
*.lzma	lzma
*.xz	xz
*.Z	compress (Lempel-Ziv)
*.zst	zstd

That's actually the only thing you need to know. There are no options.

After decompressing a file, the filetype will be detected again. This will
make a file like "foo.c.gz" get the "c" filetype.

If you have 'patchmode' set, it will be appended after the extension for
compression. Thus editing the patchmode file will not give you the automatic
decompression. You have to rename the file if you want this.

Author: Nikolay Pavlov <kp-pav@yandex.ru>
Copyright: (c) 2015 by Nikolay Pavlov

The Apache license applies to the files in this package, including
runtime/autoload/msgpack.vim, runtime/doc/pi_msgpack.txt and
test/functional/plugin/msgpack_spec.lua. Like anything else that's free,
msgpack.vim and its associated files are provided as is and comes with no
warranty of any kind, either expressed or implied. No guarantees of
merchantability. No guarantees of suitability for any purpose. By using this
plugin, you agree that in no event will the copyright holder be liable for any
damages resulting from the use of this software. Use at your own risk!

Msgpack.vim introduction

This plugin contains utility functions to be used in conjunction with
msgpackdump() and msgpackparse() functions.

Msgpack.vim manual

msgpack.vim-manual

FUNCTION ARGUMENTS

msgpack.vim-arguments

Disambiguation of arguments described below. **Note:** if e.g. function is listed
as accepting {msgpack-integer} (or anything else) it means that function
does not check whether argument matches its description.

{msgpack-value}	Either msgpack-special-dict or a regular value, but not function reference.
{msgpack-integer}	Any value for which msgpack#type() will return "integer".
{msgpack-special-int}	msgpack-special-dict representing integer.

msgpack#is_int({msgpack-value}) msgpack#is_int()

Returns 1 if given {msgpack-value} is integer value, 0 otherwise.

msgpack#is_uint({msgpack-value}) msgpack#is_uint()

Returns 1 if given {msgpack-value} is integer value greater or equal
to zero, 0 otherwise.

msgpack#strftime

msgpack#strftime({format}, {msgpack-integer}) msgpack#strftime()

Same as strftime(), but second argument may be
msgpack-special-dict. Requires Python to really work with
msgpack-special-dicts.

msgpack#strptime

msgpack#strptime({format}, {time}) msgpack#strptime()

Reverse of msgpack#strftime(): for any time and format
msgpack#equal(msgpack#strptime(format, msgpack#strftime(format,
time)), time) be true. Requires ||Python|, without it only supports
non-|msgpack-special-dict| nonnegative times and format equal to
%Y-%m-%dT%H:%M:%S.

msgpack#int_dict_to_str({msgpack-special-int}) msgpack#int_dict_to_str()
Function which converts msgpack-special-dict integer value to
a hexadecimal value like 0x1234567890ABCDEF (always returns exactly 16
hexadecimal digits).

msgpack#special_type({msgpack-value}) msgpack#special_type()

Returns zero if {msgpack-value} is not msgpack-special-dict. If it
is it returns name of the key in v:msgpack_types which represents
{msgpack-value} type.

msgpack#type({msgpack-value}) msgpack#type()

Returns name of the key in v:msgpack_types that represents
{msgpack-value} type. Never returns zero: this function returns
msgpack type which will be dumped by msgpackdump() should it receive
a list with single {msgpack-value} as input.

msgpack#deepcopy({msgpack-value}) msgpack#deepcopy()

Like deepcopy(), but works correctly with msgpack-special-dict
values. Plain deepcopy() will destroy all types in
msgpack-special-dict values because it will copy _TYPE key values,
while they should be preserved.

msgpack#string({msgpack-value}) msgpack#string()

Like string(), but saves information about msgpack types. Values
dumped by msgpack#string may be read back by msgpack#eval().
Returns is the following:

- Dictionaries are dumped as "{key1: value1, key2: value2}". **Note:**

  msgpack allows any values in keys, so with some
  msgpack-special-dict values msgpack#string() may produce even
  "{{1: 2}: 3, [4]: 5}".

- Lists are dumped as "[value1, value2]".
- Strings are dumped as

  1. "abc": binary string.
  2. ="abc": string.
  3. +(10)"ext": extension strings (10 may be replaced with any 8-bit signed integer).

  Inside strings the following escape sequences may be present: "\0" (represents NUL byte), "\n" (represents line feed) and "\"" (represents double quote).
- Floating-point and integer values are dumped using string() or msgpack#int_dict_to_str().
- Booleans are dumped as "TRUE" or "FALSE".
- Nil values are dumped as "NIL".

msgpack#eval({string}, {dict}) msgpack#eval()

Transforms string created by msgpack#string() into a value suitable for msgpackdump(). Second argument allows adding special values that start with head characters (/\h) and contain only word characters (/\w). Built-in special values are "TRUE", "FALSE", "NIL", "nan" and "inf" and they cannot be overridden. Map values are always evaluated to msgpack-special-dict values, as well as hexadecimal digits. When evaluating maps order of keys is preserved.

Note that in addition to regular integer representations that may be obtained using msgpack#string() msgpack#eval() also supports C-style “character” integer constants like '/' (equivalent to char2nr('/'): 47). This also allows '\0' (number is decimal).

msgpack#equal

msgpack#equal({msgpack-value}, {msgpack-value}) msgpack#equal()

Returns 1 if given values are equal, 0 otherwise. When comparing msgpack map values order of keys is ignored. Comparing msgpack-special-dict with equivalent non-special-dict value evaluates to 1.

Highlighting matching parens

The functionality mentioned here is a standard-plugin.
This plugin is only available if 'compatible' is not set.

You can avoid loading this plugin by setting the "loaded_matchparen" variable:

```
:let loaded_matchparen = 1
```

The plugin installs CursorMoved, CursorMovedI and WinEnter autocommands to
redefine the match highlighting.

:NoMatchParen :DoMatchParen

To disable the plugin after it was loaded use this command:

```
:NoMatchParen
```

And to enable it again:

```
:DoMatchParen
```

The highlighting used is MatchParen. You can specify different colors with
the ":highlight" command. Example:

```
:hi MatchParen ctermbg=blue guibg=lightblue
```

By default the plugin will highlight both the paren under the cursor and the
matching one using the hl-MatchParen highlighting group. This may result in
the cursor briefly disappearing from the screen as the MatchParen colors take
over the cursor highlight. To prevent this from happening and have the plugin
only highlight the matching paren and not the one under the cursor
(effectively leaving the cursor style unchanged), you can set the
"matchparen_disable_cursor_hl" variable:

```
:let matchparen_disable_cursor_hl = 1
```

The characters to be matched come from the 'matchpairs' option. You can
change the value to highlight different matches. Note that not everything is
possible. For example, you can't highlight single or double quotes, because
the start and end are equal.

The syntax highlighting attributes are used. When the cursor currently is not
in a string or comment syntax item, then matches inside string and comment
syntax items are ignored. Any syntax items with "string" or "comment"
somewhere in their name are considered string or comment items.

The search is limited to avoid a delay when moving the cursor. The limits
are:

- What is visible in the window.
- 100 lines above or below the cursor to avoid a long delay when there are closed folds.
- 'synmaxcol' times 2 bytes before or after the cursor to avoid a delay in a long line with syntax highlighting.
- A timeout of 300 msec (60 msec in Insert mode). This can be changed with the "g:matchparen_timeout" and "g:matchparen_insert_timeout" variables and their buffer-local equivalents "b:matchparen_timeout" and "b:matchparen_insert_timeout".

If you would like the % command to work better, the |matchit| plugin can be
used. This plugin also helps to skip matches in comments. This is unrelated
to the matchparen highlighting, they use a different mechanism.

by Gustavo Niemeyer

This is a filetype plugin to work with rpm spec files.

Currently, this Vim plugin allows you to easily update the %changelog section in RPM spec files. It will even create a section for you if it doesn't exist yet. If you've already inserted an entry today, it will give you the opportunity to just add a new item in today's entry. If you don't provide a format string (spec_chglog_format), it'll ask you an email address and build a format string by itself.

1. How to use it spec-how-to-use-it

2. Customizing spec-customizing

1. How to use it

The spec_chglog plugin provides a map like the following:

```
:map <buffer> <LocalLeader>c <Plug>SpecChangelog
```

It means that you may run the plugin inside a spec file by pressing your maplocalleader key (default is '\') plus 'c'. If you do not have spec_chglog_format set, the plugin will ask you for an email address to use in this edit session.

Every time you run the plugin, it will check to see if the last entry in the changelog has been written today and by you. If the entry matches, it will just insert a new changelog item, otherwise it will create a new changelog entry. If you are running with spec_chglog_release_info enabled, it will also check if the name, version and release matches. The plugin is smart enough to ask you if it should update the package release, if you have not done so.

Setting a map spec-setting-a-map

As you should know, you can easily set a map to access any Vim command (or anything, for that matter). If you don't like the default map of <LocalLeader>c, you may just set up your own key. The following line shows you how you could do this in your vimrc file, mapping the plugin to the <F5> key:

```
au FileType spec map <buffer> <F5> <Plug>SpecChangelog
```

Note: the plugin will respect your desire to change the default mapping and won't set it.

This command will add a map only in the spec file buffers.

2. Customizing spec-customizing

The format string spec_chglog_format

You can easily customize how your spec file entry will look like. To do this just set the variable "spec_chglog_format" in your vimrc file like this:

```
let spec_chglog_format = "%a %b %d %Y My Name <my@email.com>"
```

Note that "%a %b %d %Y" is the most used time format. If you don't provide a format string, when you run the SpecChangelog command for the first time, it will ask you an email address and build the spec_chglog_format variable for you. This way, you will only need to provide your email address once.

To discover which format options you can use, take a look at the strftime() function man page.

Where to insert new items spec_chglog_prepend

The plugin will usually insert new %changelog entry items (note that it's not the entry itself) after the existing ones. If you set the spec_chglog_prepend variable

```
let spec_chglog_prepend = 1
```

it will insert new items before the existing ones.

Inserting release info spec_chglog_release_info

If you want, the plugin may automatically insert release information on each changelog entry. One advantage of turning this feature on is that it may control if the release has been updated after the last change in the package or not. If you have not updated the package version or release, it will ask you if it should update the package release for you. To turn this feature on, just insert the following code in your vimrc:

```
let spec_chglog_release_info = 1
```

Then, the first item in your changelog entry will be something like:

```
+ name-1.0-1cl
```

If you don't like the release updating feature and don't want to answer
"No" each time it detects an old release, you may disable it with

```
let spec_chglog_never_increase_release = 1
```

Good luck!!

```
                    +====================+
                    | Tar File Interface |
                    +====================+
```

Original Author: Charles E. Campbell
Copyright 2005-2017:
        The VIM LICENSE (see copyright) applies to the files in this
        package, including tarPlugin.vim, tar.vim, and pi_tar.txt. Like
        anything else that's except use "tar.vim" instead of "VIM". Like
        anything else that's free, tar.vim and its associated files are
        provided as is and comes with no warranty of any kind, either
        expressed or implied. No guarantees of merchantability. No
        guarantees of suitability for any purpose. By using this plugin, you
        agree that in no event will the copyright holder be liable for any
        damages resulting from the use of this software. Use at your own risk!

1. Contents

tar tar-contents

1. Contents.....tar-contents

2. Usage.....tar-usage

3. Options.....tar-options

4. History.....tar-history

2. Usage

tar-usage tar-manual

When one edits a *.tar file, this plugin will handle displaying a
contents page. Select a file to edit by moving the cursor atop
the desired file, then hit the <return> key. After editing, one may
also write to the file. Currently, one may not make a new file in
tar archives via the plugin.

:TarDiff

DIFFERENCING SUPPORT~

:TarDiff [filename]

This command will attempt to show the differences between the tarball
version of a file and the associated file on the system. In order to
find that file on the system, the script uses the path associated with
the file mentioned in the tarball. If the current directory is not
correct for that path, :TarDiff will fail to find the associated file.

If the [filename] is given, that filename (and path) will be used to
specify the associated file.

PREVENTING LOADING~

If for some reason you do not wish to use vim to examine tar'd files,
you may put the following two variables into your <.vimrc> to prevent
the tar plugin from loading:

```
let g:loaded_tarPlugin= 1
let g:loaded_tar      = 1
```

tar-mappings

MAPPINGS~

The following (buffer-local) mappings are available in a tar buffer:

<CR>	Open selected file for editing, any changes will be written back to the archive.
<LeftMouse>	same as <CR>
x	Extract selected file.

3. Options

tar-options

These options are variables that one may change, typically in one's
<.vimrc> file.

Variable	Default Value	Explanation
g:tar_browseoptions	"Ptf"	used to get a list of contents
g:tar_readoptions	"OPxf"	used to extract a file from a tarball
g:tar_cmd	"tar"	the name of the tar program
g:tar_nomax	0	if true, file window will not be maximized
g:tar_writeoptions	"uf"	used to update/replace a file

4. History

tar-history

unreleased:
        Apr 16, 2026   * add missing g:tar_secure into tar#Extract
        Apr 15, 2026   * add path traversal checks in tar#Extract()
        Apr 09, 2026   * fix zstd support and dotted filename support
        Apr 06, 2026   * fix lz4 support

```
        Feb 07, 2026    * make path traversal detection more robust
        Feb 06, 2026    * fix bug with nowrapscan
        Jul 16, 2025    * update minimum required Vim version
        Jul 13, 2025    * drop leading /
        May 19, 2025    * restore working directory after read/write
        Apr 16, 2025    * decouple from netrw by adding s:WinPath()
                          instead of shelling out to file(1)
        Mar 02, 2025    * determine the compression using readblob()
        Mar 02, 2025    * escape the filename before using :read
        Mar 01, 2025    * fix syntax error in tar#Read()
        Feb 28, 2025    * add support for bzip3 (#16755)
        Feb 06, 2025    * add support for lz4 (#16591)
        Nov 11, 2024    * support permissions (#7379)
        Feb 19, 2024    * announce adoption
        Jan 08, 2024    * fix a few problems (#138331, #12637, #8109)
v31     Apr 02, 2017    * (klartext) reported that browsing encrypted
                          files in a zip archive created unencrypted
                          swap files.  I am applying a similar fix
                          used on zip.vim to tar.vim: new buffers
                          are opened with :noswapfile.
        May 16, 2017    * When the mouse option isn't empty, the
                          leftmouse can be used to select a file
                          in the tar-file listing.
v30     Apr 22, 2014    * .tgz files are ambiguous: they may have been
                          compressed with either gzip or bzip2.  Tar.vim
                          disambiguates by using unix's "file" command.
        Feb 18, 2016    * Changed =~ to =~# where appropriate
        Feb 18, 2017    * Now also permits xz decompression
v28     Jun 23, 2011    * a few more decompression options (tbz tb2 txz)
v27     May 31, 2011    * moved cygwin detection before g:tar_copycmd
                          handling
                        * inserted additional :keepj modifiers
                        * changed silent  to  sil!  (:silent)
v26     Aug 09, 2010    * uses buffer-local instead of window variables
                          to hold tarfile name
                        * inserted keepj before 0d to protect jump list
v25     Jun 19, 2010    * (Jan Steffens) added support for xz
                          compression
v24     Apr 07, 2009    * :Untarvim command implemented
        Sep 28, 2009    * Added lzma support
v22     Aug 08, 2008    * security fixes
v16     Jun 06, 2008    * tarfile:: used instead of tarfile: when
                          editing files inside tarballs.  Fixes a
                          problem with tarballs called things like
                          c:\abc.tar. (tnx to Bill McCarthy)
v14     May 09, 2008    * arno caught a security bug
        May 28, 2008    * various security improvements.  Now requires
                          patch 299 which provides the fnameescape()
                          function
        May 30, 2008    * allows one to view *.gz and *.bz2 files that
                          are in *.tar files.
v12     Sep 07, 2007    * &shq now used if not the empty string for
                          g:tar_shq
v10     May 02, 2006    * now using "redraw then echo" to show messages,
                          instead of "echo and prompt user"
v9      May 02, 2006    * improved detection of masquerading as tar file
v8      May 02, 2006    * allows editing of files that merely masquerade
                          as tar files
v7      Mar 22, 2006    * work on making tar plugin work across network
        Mar 27, 2006    * g:tar_cmd now available for users to change
                          the name of the tar program to be used.  By
                          default, of course, it's "tar".
v6      Dec 21, 2005    * writing to files not in directories caused
                          problems - fixed (pointed out by
                          Christian Robinson)
v5      Nov 22, 2005    * report option workaround installed
v3      Sep 16, 2005    * handles writing files in an archive back to
                          the archive
        Oct 18, 2005    * <amatch> used instead of <afile> in autocmds
        Oct 18, 2005    * handles writing to compressed archives
        Nov 03, 2005    * handles writing tarfiles across a network
                          using netrw#NetWrite()
v2                      * converted to use Vim7's new autoload feature
                          by Bram Moolenaar
v1      (original)      * Michael Toren
                          (see michael.toren.net/code/
                           link seems dead)
```

vim-tutor-mode provides a system to follow and create interactive tutorials
for vim and third party plugins. It replaces the venerable vimtutor system.

Original Author: Felipe Morales <https://github.com/fmoralesc>

1. Usage

vim-tutor-usage

vim-tutor-mode tutorials are hypertext documents, they have rich text and
contain links. To stand out from the rest of the text, links are underlined.
You can follow them by placing the cursor over them and pressing <Enter>, or
by double-clicking them.

1.1 Commands

:Tutor {tutorial}

Opens a tutorial. Command-line completion for {tutorial} is provided, the candidates are a list of '.tutor' files found in the 'tutor/' folder in the 'runtimepath'. Tutorials prefixed with 'vim-' will always be shown first.

If no {tutorial} is provided, the command starts the 'vim-01-beginner' tutorial, which is equivalent to Vim's vimtutor.

2. Creating tutorials

vim-tutor-create

Writing vim-tutor-mode tutorials is easy. For an overview of the format used,
please consult the 'tutor.tutor' file:

```
:Tutor tutor
```

New tutorials must be placed in the 'tutor/' folder in the 'runtimepath'
to be detected by the :Tutor command.

It is recommended to use a less formal style when writing tutorials than in
regular documentation (unless the content requires it).

Plugins and modules included with Nvim

Nvim includes various Lua and Vim plugins or modules which may provide commands (such as :TOhtml) or modules that you can optionally require() or :packadd. The Lua ones are not part of the lua-stdlib, that is, they are not available from the global vim module namespace. Some of the plugins are loaded by default while others are not loaded until requested by :packadd.

Standard plugins

standard-plugin-list

Help-link	Loaded	Short description
dir	Yes	Directory listing for :edit
difftool	No	Compares directories (and files) side-by-side
editorconfig	Yes	Detect and interpret editorconfig
ft-shada	Yes	Allows editing binary shada files
man.lua	Yes	View manpages in Nvim
|matchit|	Yes	Extended % matching
matchparen	Yes	Highlight matching pairs
netrw	No	Reading and writing files over a network
zip	Yes	Read-only zip archive browser
package-cfilter	No	Filtering quickfix/location list
package-justify	No	Justify text
package-nohlsearch	No	Automatically run :nohlsearch
package-spellfile	Yes	Install spellfile if missing
package-termdebug	No	Debug inside Nvim with gdb
package-tohtml	No	Convert buffer to html, syntax included
package-undotree	No	Interactive textual undotree
pi_gzip.txt	Yes	Reading and writing compressed files
pi_msgpack.txt	No	msgpack utilities
pi_paren.txt	Yes	Highlight matching parens
pi_spec.txt	Yes	Filetype plugin to work with rpm spec files
pi_swapmouse	No	Swap meaning of left and right mouse buttons
pi_tar.txt	Yes	Tar file explorer
pi_tutor.txt	Yes	Interactive tutorial
old-zip	No	Legacy zip archive explorer

Builtin plugin: dir

dir

Nvim opens a directory listing when you :edit a directory path, handled by the builtin "dir" plugin. The listing is read-only, dir.lua does not provide actions which modify the filesystem.

Each "dir" buffer is initialized as follows:

- Sets 'filetype' to "directory".
- Sets current-directory via :bcd, so gf, :! and friends work as expected.
- Keeps alternate-file, so CTRL-^ returns to the buffer you came from.

g:loaded_nvim_dir_plugin

To disable the built-in directory browser, set this before startup:

```
vim.g.loaded_nvim_dir_plugin = 1
```

GLOBAL MAPPINGS

dir-mappings

- - opens the parent directory of the current file or directory.
- {count}- is like -, but a count of 1 opens the global working directory, and a higher count goes up that many levels.

BUFFER-LOCAL MAPPINGS

dir-buffer-mappings

- <CR> opens the file or directory under the cursor.
- a opens the file or directory under the cursor in a vertical split.

- o opens the file or directory under the cursor in a horizontal split.
- - opens the parent directory.
- R reloads the directory listing.

a and o are available for local directories and keep focus in the listing.
'splitright' and 'splitbelow' control where the new window appears.

These keys map to <Plug>(nvim-dir-open), <Plug>(nvim-dir-vsplit),
<Plug>(nvim-dir-split), <Plug>(nvim-dir-up), and <Plug>(nvim-dir-reload); map those
to use different keys. Default mappings are skipped when the target <Plug>
mapping is already mapped. The default "-", "a", and "o" mappings are also
skipped when their keys are already mapped.

dir-config

Directory buffers follow the global 'hidden' option by default. To delete them
after use:

```
autocmd FileType directory setlocal bufhidden=delete
```

A discarded listing is rebuilt on the next visit, so the cursor starts at the
first entry instead of the one it was left on. "wipe" discards the buffer
itself, leaving no alternate-file; see 'bufhidden'.

Reshaping the listing dir-render

DirReadPost

A User autocommand fired after each listing is rendered, including reloads.
The buffer is writable for the duration, so a handler can sort or filter it
with ordinary commands, and the cursor is placed afterwards.

Handlers may reorder or remove lines, but each remaining line must still be an
entry name: dir-buffer-mappings resolve the line under the cursor against the
buffer name, so rewriting line text makes <CR> open a nonexistent path.

Sort directories last:

```
autocmd User DirReadPost silent keeppatterns sort r /\/$/
```

Hide dot-prefixed entries:

```
autocmd User DirReadPost silent keeppatterns g/^\./d _
```

Sort by modification time, newest first:

```
vim.api.nvim_create_autocmd('User', {
  pattern = 'DirReadPost',
  callback = function(args)
    local dir = vim.api.nvim_buf_get_name(args.buf)
    local names = vim.api.nvim_buf_get_lines(args.buf, 0, -1, true)
    local mtime = {} --- @type table<string, integer>
    for _, name in ipairs(names) do
      local stat = vim.uv.fs_stat(vim.fs.joinpath(dir, name))
      mtime[name] = stat and stat.mtime.sec or 0
    end
    table.sort(names, function(a, b)
      return mtime[a] > mtime[b]
    end)
    vim.api.nvim_buf_set_lines(args.buf, 0, -1, true, names)
  end,
})
```

Decorating the listing dir-decorate

A nvim_set_decoration_provider() sets extmarks on the visible lines of
each redraw, so they survive reordering by a DirReadPost handler.

Classify each entry, and show where a symbolic link points:

```
local ns = vim.api.nvim_create_namespace('my.dir.classify')
local glyph = {
  fifo = '|', socket = '=', char = '%', block = '#',
}
vim.api.nvim_set_decoration_provider(ns, {
  on_win = function(_, _, buf)
    return vim.bo[buf].filetype == 'directory'
  end,
  on_range = function(_, _, buf, row)
    local dir = vim.api.nvim_buf_get_name(buf)
    local name = vim.api.nvim_buf_get_lines(buf, row, row + 1, true)[1]
    local path = vim.fs.joinpath(dir, (name:gsub('/$', '')))
    local stat = vim.uv.fs_lstat(path) or {}
    local exe = stat.type == 'file'
      and bit.band(stat.mode, tonumber('111', 8)) ~= 0
    local char = glyph[stat.type] or (exe and '*')
    if char then
      vim.api.nvim_buf_set_extmark(buf, ns, row, #name, {
        virt_text = { { char, 'Dimmed' } },
        virt_text_pos = 'overlay',
        ephemeral = true,
      })
    end
    if stat.type == 'link' then
      local target = vim.uv.fs_readlink(path) or '?'
      vim.api.nvim_buf_set_extmark(buf, ns, row, 0, {
        virt_text = { { '-> ' .. target, 'Dimmed' } },
        virt_text_pos = 'eol',
        ephemeral = true,
      })
    end
    return row + 1
  end,
})
```

Replacing the directory browser dir-disable

To use another directory browser for the current session, delete the
nvim.dir autocommand group on or after VimEnter and handle FileType "directory".
This clears the built-in directory-opening autocommands.

```
vim.api.nvim_del_augroup_by_name('nvim.dir')
vim.api.nvim_create_autocmd('FileType', {
  pattern = 'directory',
  callback = function(args)
    require('my_browser').open(args.buf, vim.api.nvim_buf_get_name(args.buf))
  end,
})
```

netrw

To enable the legacy "netrw" plugin:

```
:packadd netrw
```

The plugin and its mappings remain loaded (it is a core component used by
other features such as zip). Define replacement keymaps explicitly, for
example:

```
vim.keymap.set('n', '-', function()
  require('my_browser').open_parent()
end)
```

Builtin plugin: difftool difftool

:DiffTool {left} {right} :DiffTool

Compares two directories or files side-by-side.
Supports directory diffing, rename detection, and highlights changes
in quickfix list. Replaces the built-in nvim -d diff mode with this interface.

The plugin is not loaded by default; use :packadd to activate it:

```
:packadd nvim.difftool
```

Example git difftool -d integration using nvim -d replacement:

```
[difftool "nvim_difftool"]
  cmd = nvim -c \"packadd nvim.difftool\" -d \"$LOCAL\" \"$REMOTE\"
[diff]
  tool = nvim_difftool
```

open({left}, {right}, {opt}) difftool.open()

Diff two files or directories

Parameters:

- {left}   (string)
- {right} (string)
- {opt}   (table?)
    - {rename.detect} (boolean, default: false) Whether to
      detect renames
    - {rename.similarity} (number, default: 0.5) Minimum
      similarity for rename detection (0 to 1)
    - {rename.chunk_size} (number, default: 4096) Maximum
      chunk size to read from files for similarity calculation
    - {ignore} (string[], default: {}) List of file patterns
      to ignore (for example: '.git', '*.log')
    - {method} ('auto'|'builtin'|'diffr', default: auto) Diff
      method to use
    - {rename} (table) Controls rename detection

Builtin plugin: editorconfig

editorconfig

EditorConfig is like 'modeline' for an entire (recursive) directory. When a file is opened, after running ftplugins and FileType autocommands, the EditorConfig feature searches all parent directories of that file for .editorconfig files, parses them, and applies their properties. For more information see https://editorconfig.org/.

Example .editorconfig file:

```
root = true

[*]
charset = utf-8
end_of_line = lf
indent_size = 4
indent_style = space
max_line_length = 42
trim_trailing_whitespace = true

[*.{diff,md}]
trim_trailing_whitespace = false
```

g:editorconfig b:editorconfig

EditorConfig is enabled by default. To disable it, add to your config:

```
vim.g.editorconfig = false
```

(Vimscript: let g:editorconfig = v:false). It can also be disabled per-buffer by setting the b:editorconfig buffer-local variable to false.

Nvim stores the applied properties in b:editorconfig if it is not false.

editorconfig-custom-properties

New properties can be added by adding a new entry to the "properties" table. The table key is a property name and the value is a callback function which accepts the number of the buffer to be modified, the value of the property in the .editorconfig file, and (optionally) a table containing all of the other properties and their values (useful for properties which depend on other properties). The value is always a string and must be coerced if necessary. Example:

```
require('editorconfig').properties.foo = function(bufnr, val, opts)
  if opts.charset and opts.charset ~= "utf-8" then
    error("foo can only be set when charset is utf-8", 0)
  end
  vim.b[bufnr].foo = val
end
```

editorconfig-properties

The following properties are supported by default:

charset editorconfig.charset

One of "utf-8", "utf-8-bom", "latin1", "utf-16be", or "utf-16le". Sets the 'fileencoding' and 'bomb' options.

end_of_line editorconfig.end_of_line

One of "lf", "crlf", or "cr". These correspond to setting 'fileformat' to "unix", "dos", or "mac", respectively.

indent_size editorconfig.indent_size

A number indicating the size of a single indent. Alternatively, use the value "tab" to use the value of the tab_width property. Sets the 'shiftwidth' and 'softtabstop' options. If this value is not "tab" and the tab_width property is not set, 'tabstop' is also set to this value.

indent_style editorconfig.indent_style

One of "tab" or "space". Sets the 'expandtab' option.

insert_final_newline editorconfig.insert_final_newline

"true" or "false" to ensure the file always has a trailing newline as its last byte. Sets the 'fixendofline' and 'endofline' options.

max_line_length editorconfig.max_line_length

A number indicating the maximum length of a single line. Sets the 'textwidth' option.

root editorconfig.root

If "true", then stop searching for .editorconfig files in parent directories. This property must be at the top-level of the .editorconfig file (i.e. it must not be within a glob section). It only controls the search and is excluded from property callbacks and b:editorconfig.

spelling_language editorconfig.spelling_language

A code of the format ss or ss-TT, where ss is an ISO 639 language code and TT is an ISO 3166 territory identifier. Sets the 'spelllang' option.

tab_width editorconfig.tab_width

The display size of a single tab character. Sets the 'tabstop' option.

trim_trailing_whitespace editorconfig.trim_trailing_whitespace

When "true", trailing whitespace is automatically removed when the buffer is written.

Builtin plugin: spellfile package-spellfile

Asks the user to download missing spellfiles. The spellfile is written to stdpath('data') .. 'site/spell' or the first writable directory in the 'runtimepath'.

The plugin can be disabled by setting g:loaded_spellfile_plugin = 1.

nvim.spellfile.Config
A table with the following fields:

Fields:

- {confirm} (boolean, default: true) Whether to ask user to confirm download.
- {timeout_ms} (integer, default: 15000) Number of milliseconds after which the vim.net.request() times out.
- {url} (string) The base URL from where the spellfiles are downloaded. Uses g:spellfile_URL if it's set, otherwise https://ftp.nluug.nl/pub/vim/runtime/spell.

config({opts}) spellfile.config()

Configure spellfile download options. Omit {opts} to retrieve the current configuration. Otherwise, update only the supplied fields.

For example:

```
require('nvim.spellfile').config({ url = '...' })
```

Parameters:

- {opts} (Partial<nvim.spellfile.Config>?)

Return:
(nvim.spellfile.Config?) Current config if {opts} is omitted.

get({lang}) spellfile.get()

Download spellfiles for language {lang} if available.

Parameters:

- {lang} (string) Language code.

Return:
(table?) A table with the following fields:

- {dir} (string)
- {encoding} (string)
- {files} (string[])
- {key} (string)
- {lang} (string)

Builtin plugin: tohtml

package-tohtml

:[range]TOhtml {file}

:TOhtml

Converts the buffer shown in the current window to HTML, opens the generated HTML in a new split window, and saves its contents to {file}. If {file} is not given, a temporary file (created by tempname()) is used.

The plugin is not loaded by default; use :packadd to activate it:

```
:packadd nvim.tohtml
```

tohtml({winid}, {opt})

tohtml.tohtml()

Converts the buffer shown in the window {winid} to HTML and returns the output as a list of string.

Parameters:

- {winid} (integer?) Window to convert (defaults to current window)
- {opt} (table?) Optional parameters.
  - {font}? (string[]|string, default: guifont) Fonts to use.
  - {number_lines}? (boolean, default: false) Show line numbers.
  - {range}? (integer[], default: entire buffer) Range of rows to use.
  - {title}? (string|false, default: buffer name) Title tag to set in the generated HTML code.
  - {width}? (integer, default: 'textwidth' if non-zero or window width otherwise) Width used for items which are either right aligned or repeat a character infinitely.

Return:
(string[])

Builtin plugin: undotree

package-undotree

open({opts})

undotree.open()

Open a window that displays a textual representation of the undo-tree, or closes the window if it is already open. Can also be shown with :Undotree.

:Undotree

While in the window, moving the cursor changes the undo.

The plugin is not loaded by default; use :packadd to activate it:

```
:packadd nvim.undotree
```

Parameters:

- {opts} (table?) A table with the following fields:
  - {bufnr} (integer?) Buffer to draw the tree into. If omitted, a new buffer is created.
  - {command} (string?) Vimscript command to create the

window. Default value is "30vnew". Only used when {winid} is nil.

- {title} ((string|fun(bufnr:integer):string?)?) Title of the window. If a function, it accepts the buffer number of the source buffer as its only argument and should return a string.
- {winid} (integer?) Window id to display the tree buffer in. If omitted, a new window is created with {command}.

Return:

(boolean?) Returns true if the window was already open, nil otherwise

Providers

Nvim delegates some features to dynamic "providers". This document describes the providers and how to install them. E319
Use of a feature requiring a missing provider is an error:

```
E319: No "foo" provider found. Run ":checkhealth vim.provider"
```

Run the :checkhealth command, and review the sections below.

Python integration provider-python

Nvim supports Python remote-plugins and the Vim legacy python3 and pythonx interfaces (which are implemented as remote-plugins).

Note: Only the Vim 7.3 legacy interface is supported, not later features such as python-bindeval (Vim 7.4); use the Nvim API instead. Python 2 is not supported.

PYTHON QUICKSTART

To use Python plugins, you need the "pynvim" module. Run :checkhealth to see if you already have it (some package managers install the module with Nvim itself).

For Python 3 plugins:

1. Make sure Python 3.9+ is available in your $PATH.
2. Install either uv (https://docs.astral.sh/uv/) or pipx (https://pipx.pypa.io/stable/).
3. Install the module:

```
pipx install pynvim
# or:
uv tool install --upgrade pynvim
```

The `--upgrade` flag ensures that you get the latest version even if a previous version was already installed.

See also python-virtualenv.

PYTHON PROVIDER CONFIGURATION

g:python3_host_prog

Command to start Python 3 (executable, not directory). Setting this makes startup faster. Useful for working with virtualenvs. Must be set before any check for has("python3").

```
let g:python3_host_prog = '/path/to/python3'
```

g:loaded_python3_provider

To disable Python 3 support:

```
let g:loaded_python3_provider = 0
```

PYTHON VIRTUALENVS

python-virtualenv

Using pynvim 0.6.0+ installed via uv or pipx, Nvim will automatically detect pynvim even if other Python virtual environments are activated (technical note: via the "pynvim-python" global python tool). For older pynvim (or older Neovim), where detection involved finding the first Python interpreter and checking if it could import pynvim, automatic detection would fail when another virtual environment is active. Upgrading to the latest pynvim is the recommended solution to this; but if that's not an option, then you can set the variable g:python3_host_prog in `init.vim` to point to the full path to the Python interpreter where `pynvim` is installed, e.g.:

```
let g:python3_host_prog = '/path/to/pynvim-venv/bin/python'
```

See also: https://github.com/zchee/deoplete-jedi/wiki/Setting-up-Python-for-Neovim

Ruby integration provider-ruby

Nvim supports Ruby remote-plugins and the Vim legacy ruby-vim interface (which is itself implemented as a Nvim remote-plugin).

RUBY QUICKSTART

To use Ruby plugins with Nvim, install the latest "neovim" RubyGem:

```
gem install neovim
```

Run :checkhealth to see if your system is up-to-date.

RUBY PROVIDER CONFIGURATION

g:loaded_ruby_provider

To disable Ruby support:

```
let g:loaded_ruby_provider = 0
```

g:ruby_host_prog

Command to start the Ruby host. By default this is "neovim-ruby-host". With project-local Ruby versions (via tools like RVM or rbenv) setting this can avoid the need to install the "neovim" gem in every project.

To use an absolute path (e.g. to an rbenv installation):

```
let g:ruby_host_prog = '~/.rbenv/versions/2.4.1/bin/neovim-ruby-host'
```

To use the RVM "system" Ruby installation:

```
let g:ruby_host_prog = 'rvm system do neovim-ruby-host'
```

Perl integration

provider-perl

Nvim supports Perl remote-plugins on Unix platforms. Support for polling STDIN on MS-Windows is currently lacking from all known event loop implementations. The Vim legacy perl-vim interface is also supported (which is itself implemented as a Nvim remote-plugin). https://github.com/jacquesg/p5-Neovim-Ext

Note: Only perl versions from 5.22 onward are supported.

PERL QUICKSTART

To use perl remote-plugins with Nvim, install the "Neovim::Ext" cpan package:

```
cpanm -n Neovim::Ext
```

Run :checkhealth to see if your system is up-to-date.

PERL PROVIDER CONFIGURATION

g:loaded_perl_provider

To disable Perl support:

```
:let g:loaded_perl_provider = 0
```

g:perl_host_prog

Command to start the Perl executable. Must be set before any check for has("perl").

```
let g:perl_host_prog = '/path/to/perl'
```

Node.js integration

provider-nodejs

Nvim supports Node.js remote-plugins. https://github.com/neovim/node-client/

NODEJS QUICKSTART

To use javascript remote-plugins with Nvim, install the "neovim" npm package:

```
npm install -g neovim
```

Run :checkhealth to see if your system is up-to-date.

NODEJS PROVIDER CONFIGURATION

g:loaded_node_provider

To disable Node.js support:

```
:let g:loaded_node_provider = 0
```

g:node_host_prog

Command to start the Node.js host. Setting this makes startup faster.

By default, Nvim searches for "neovim-node-host" using "npm root -g", which can be slow. To avoid this, set g:node_host_prog to the host path:

```
let g:node_host_prog = '/usr/local/bin/neovim-node-host'
```

Clipboard integration

provider-clipboard clipboard

Nvim has no direct connection to the system clipboard. Instead it depends on a provider which transparently uses shell commands to communicate with the system clipboard or any other clipboard "backend".

To ALWAYS use the clipboard for ALL operations (instead of interacting with the "+" and/or "*" registers explicitly):

```
set clipboard+=unnamedplus
```

See 'clipboard' for details and options.

clipboard-tool

The presence of a working clipboard tool implicitly enables the "+" and "*" registers. Nvim supports these clipboard tools, in order of priority:

- g:clipboard : User override (if set to a dict or any string "name" below; e.g. `g:clipboard="tmux"` forces tmux clipboard and skips auto-detection).
- "pbcopy" : pbcopy, pbpaste (macOS)
- "wl-copy" : wl-copy, wl-paste (if $WAYLAND_DISPLAY is set)
- "wayclip" : waycopy, waypaste (if $WAYLAND_DISPLAY is set)
- "xsel" : xsel (if $DISPLAY is set)
- "xclip" : xclip (if $DISPLAY is set)
- "lemonade" : lemonade (for SSH) https://github.com/pocke/lemonade
- "doitclient": doitclient (for SSH) https://www.chiark.greenend.org.uk/~sgtatham/doit/
- "win32yank" : win32yank (Windows)
- "putclip" : putclip, getclip (Windows) https://cygwin.com/packages/summary/cygutils.html
- "clip" : clip, powershell (Windows) https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/clip
- "termux" : termux (via termux-clipboard-set, termux-clipboard-get)
- "tmux" : tmux (if $TMUX is set)
- "osc52" : clipboard-osc52 (if supported by your terminal)

g:clipboard

To configure a custom clipboard tool, set `g:clipboard` to a string name (from the above clipboard-tool list), or dict (to explicitly specify the shell commands or lambda functions) before clipboard providers are initialized.

Note: Clipboard providers are initialized by calling `has('clipboard')`, which is best avoided before setting `g:clipboard`. See "G:CLIPBOARD SETTINGS ARE NOT USED" in faq-runtime for workarounds.

If "cache_enabled" is TRUE then when a selection is copied Nvim will cache the selection until the copy command process dies. When pasting, if the copy process has not died the cached selection is applied.

The "copy" function stores a list of lines and the register type. The "paste" function returns the clipboard as a `[lines, regtype]` list, where `lines` is a list of lines and `regtype` is a register type conforming to setreg().

Example: set to "osc52" to force OSC52, skipping auto-detection of terminal support:

```
let g:clipboard = 'osc52'
```

Example: set to "wayclip" to force waycopy/waypaste:

```
let g:clipboard = 'wayclip'
```

Example: set to a dict which integrates the tmux clipboard:

```
let g:clipboard = {
  \   'name': 'myClipboard',
  \   'copy': {
  \      '+': ['tmux', 'load-buffer', '-'],
  \      '*': ['tmux', 'load-buffer', '-'],
  \    },
  \   'paste': {
  \      '+': ['tmux', 'save-buffer', '-'],
  \      '*': ['tmux', 'save-buffer', '-'],
  \   },
  \   'cache_enabled': 1,
  \ }
```

Example: set to a dict which uses g:foo as a fake clipboard:

```
let g:clipboard = {
  \   'name': 'myClipboard',
  \   'copy': {
  \      '+': {lines, regtype -> extend(g:, {'foo': [lines, regtype]}) },
  \      '*': {lines, regtype -> extend(g:, {'foo': [lines, regtype]}) },
  \    },
  \   'paste': {
  \      '+': {-> get(g:, 'foo', [])},
  \      '*': {-> get(g:, 'foo', [])},
  \   },
  \ }
```

clipboard-wsl

For Windows WSL, try this g:clipboard definition:

```
let g:clipboard = {
  \   'name': 'WslClipboard',
  \   'copy': {
  \      '+': 'clip.exe',
  \      '*': 'clip.exe',
  \    },
  \   'paste': {
  \      '+': 'powershell.exe -NoLogo -NoProfile -c [Console]::Out.Write($(Get-Clipboard -Raw).tostring().replace("`r", ""))',
  \      '*': 'powershell.exe -NoLogo -NoProfile -c [Console]::Out.Write($(Get-Clipboard -Raw).tostring().replace("`r", ""))',
  \   },
  \   'cache_enabled': 0,
  \ }
```

clipboard-osc52

Nvim bundles a clipboard provider that allows copying to the system clipboard using OSC 52, an "Operating System Command" control-sequence that causes the terminal emulator to write to or read from the system clipboard.

When Nvim is running in the TUI, it automatically detects host terminal support for OSC 52. If successful, then Nvim will use OSC 52 for copying and pasting if no other clipboard-tool is found and when 'clipboard' is unset. **NOTE:** Using a terminal multiplexer (e.g. tmux) may inhibit automatic OSC 52 support detection.

g:termfeatures

To disable the automatic detection, set the "osc52" key of g:termfeatures to false early in your config. Example:

```
local termfeatures = vim.g.termfeatures or {}
termfeatures.osc52 = false
vim.g.termfeatures = termfeatures
```

To force Nvim to use the OSC 52 provider you can set g:clipboard:

```
vim.g.clipboard = 'osc52'
```

Which is equivalent to:

```
vim.g.clipboard = {
  name = 'OSC 52',
  copy = {
    ['+'] = require('vim.ui.clipboard.osc52').copy('+'),
    ['*'] = require('vim.ui.clipboard.osc52').copy('*'),
  },
  paste = {
    ['+'] = require('vim.ui.clipboard.osc52').paste('+'),
    ['*'] = require('vim.ui.clipboard.osc52').paste('*'),
  },
}
```

Note: not all terminal emulators support reading from the system clipboard (and even for those that do, users should be aware of the security implications), so using OSC 52 for pasting may not be possible (and not necessary, because you can paste instead using your system paste function). Users may need to configure their terminal emulator to allow reading from the clipboard.

Paste

provider-paste **paste**

"Paste" is a separate concept from clipboard: paste means "dump a bunch of text to the editor", whereas clipboard provides features like quote+ to get and set the OS clipboard directly. For example, middle-click or CTRL-SHIFT-v (macOS: CMD-v) in your terminal is "paste", not "clipboard": the terminal application (Nvim) just gets a stream of text, it does not interact with the clipboard directly.

bracketed-paste-mode

Pasting in the TUI depends on the "bracketed paste" terminal capability, which allows terminal applications to distinguish between user input and pasted text. https://cirw.in/blog/bracketed-paste This works automatically if your terminal supports it.

ui-paste

GUIs can paste by calling nvim_paste().

PASTE BEHAVIOR

Paste inserts text after the cursor. Lines break at <NL>, <CR>, and <CR><NL>. When pasting a huge amount of text, screen-updates are throttled and the message area shows a "..." pulse.

In cmdline-mode only the first line is pasted, to avoid accidentally executing many commands. Use the cmdline-window if you really want to paste multiple lines to the cmdline.

You can implement a custom paste handler by redefining vim.paste(). Example:

```
vim.paste = (function(lines, phase)
  vim.api.nvim_put(lines, 'c', true, true)
end)
```

X11 selection mechanism

clipboard-x11 **x11-selection**

X11 clipboard providers store text in "selections". Selections are owned by an application, so when the application gets closed, the selection text is lost. The contents of selections are held by the originating application (e.g., upon a copy), and only passed to another application when that other application requests them (e.g., upon a paste).

primary-selection quotestar quoteplus quote+

There are three documented X11 selections: PRIMARY, SECONDARY, and CLIPBOARD. CLIPBOARD is typically used in X11 applications for copy/paste operations (CTRL-c/CTRL-v), while PRIMARY is used for the last selected text, which is generally inserted with the middle mouse button.

Nvim's X11 clipboard providers only use the PRIMARY and CLIPBOARD selections, for the "*" and "+" registers, respectively.

This subject is introduced in section 30.1 of the user manual.

1. Using QuickFix commands

Quickfix E42

Vim has a special mode to speedup the edit-compile-edit cycle. This is inspired by the quickfix option of the Manx's Aztec C compiler on the Amiga. The idea is to save the error messages from the compiler in a file and use Vim to jump to the errors one by one. You can examine each problem and fix it, without having to remember all the error messages.

In Vim the quickfix commands are used more generally to find a list of positions in files. For example, :vimgrep finds pattern matches. You can use the positions in a script with the getqflist() function. Thus you can do a lot more than the edit/compile/fix cycle!

If you have the error messages in a file you can start Vim with:

```
vim -q filename
```

From inside Vim an easy way to run a command and handle the output is with the :make command (see below).

The 'errorformat' option should be set to match the error messages from your compiler (see errorformat below). Vim uses it to extract information such as the file name, line number and message from each line of compiler output. Lines that do not match are kept as invalid entries and are skipped by commands such as :cnext. Use :clist! to display them as well.

quickfix-stack quickfix-ID

Each quickfix list has a unique identifier called the quickfix ID and this number will not change within a Vim session. The getqflist() function can be used to get the identifier assigned to a list. There is also a quickfix list number, which is its position in the quickfix stack. This number may change when lists are added or removed, or when more than 'chistory' lists are kept. Use the ID when a script needs to refer to the same list reliably; use the list number when referring to its current position in the stack.

location-list E776

A location list is a quickfix list tied to one window. Each window can have a separate location list, and commands such as :lvimgrep, :lgrep, :lhelpgrep and :lmake use it instead of the global quickfix list. The corresponding commands without the "l" prefix use the quickfix list.

location-list-file-window

The location list is independent of the quickfix list. A location list can be associated with only one window.

When a window with a location list is split, the new window gets a copy of the location list. When there are no longer any references to a location list, the location list is destroyed.

quickfix-changedtick

Every quickfix and location list has a read-only changedtick variable that tracks the total number of changes made to the list. Every time the quickfix list is modified, this count is incremented. This can be used to perform an action only when the list has changed. The getqflist() and getloclist() functions can be used to query the current value of changedtick. You cannot change the changedtick variable.

The following quickfix commands can be used. The location list commands are similar to the quickfix commands, replacing the 'c' prefix in the quickfix command with 'l'.

E924

If the current window was closed by an autocommand while processing a location list command, it will be aborted.

E925 E926

If the current quickfix or location list was changed by an autocommand while processing a quickfix or location list command, it will be aborted.

:cc

:cc[!] [nr] Display error [nr]. If [nr] is omitted, the same
:[nr]cc[!] error is displayed again. Without [!] this doesn't work when jumping to another buffer, the current buffer has been changed, there is the only window for the buffer and both 'hidden' and 'autowrite' are off. When jumping to another buffer with [!] any changes to the current buffer are lost, unless 'hidden' is set or there is another window for this buffer. The 'switchbuf' settings are respected when jumping to a buffer. When used in the quickfix window the line number can be used, including "." for the current line and "$" for the last line.

:ll

:ll[!] [nr] Same as ":cc", except the location list for the
:[nr]ll[!] current window is used instead of the quickfix list.

:cn :cne :cnext E553
:[count]cn[ext][!] Display the [count] next error in the list that includes a file name. Entries without a file name are skipped, since Vim cannot use them to jump to a file. If there are no file names at all, go to the [count] next error. See :cc for [!] and 'switchbuf'.

]q

]q Mapped to :cnext. default-mappings

:[count]cN[ext][!] :cp :cprevious :cprev :cN :cNext
:[count]cp[revious][!] Display the [count] previous error in the list that includes a file name. Entries without a file name are skipped, since Vim cannot use them to jump to a file. If there are no file names at all, go to the [count] previous error. See :cc for [!] and 'switchbuf'.

[q

[q Mapped to :cprevious. default-mappings

:lne :lnext
:[count]lne[xt][!] Same as ":cnext", except the location list for the current window is used instead of the quickfix list.

]l

]l Mapped to :lnext. default-mappings

:[count]lN[ext][!] :lp :lprevious :lprev :lN :lNext
:[count]lp[revious][!] Same as ":cNext" and ":cprevious", except the location list for the current window is used instead of the quickfix list.

[l

[l Mapped to :lprevious. default-mappings

:cabo :cabove
:[count]cabo[ve] Go to the [count] error above the current line in the current buffer. If [count] is omitted, then 1 is used. If there are no errors, then an error message is displayed. Assumes that the entries in a quickfix list are sorted by their buffer number and line number. If there are multiple errors on the same line, then only the first entry is used. If [count] exceeds the number of entries above the current line, then the first error in the file is selected.

:lab :labove
:[count]lab[ove] Same as ":cabove", except the location list for the current window is used instead of the quickfix list.

:cbel :cbelow
:[count]cbel[ow] Go to the [count] error below the current line in the current buffer. If [count] is omitted, then 1 is used. If there are no errors, then an error message is displayed. Assumes that the entries in a quickfix list are sorted by their buffer number and line number. If there are multiple errors on the same line, then only the first entry is used. If [count] exceeds the number of entries below the current line, then the last error in the file is selected.

:lbel :lbelow
:[count]lbel[ow] Same as ":cbelow", except the location list for the current window is used instead of the quickfix list.

:cbe :cbefore
:[count]cbe[fore] Go to the [count] error before the current cursor position in the current buffer. If [count] is omitted, then 1 is used. If there are no errors, then an error message is displayed. Assumes that the entries in a quickfix list are sorted by their buffer, line and column numbers. If [count] exceeds the number of entries before the current position, then the first error in the file is selected.

:lbe :lbefore
:[count]lbe[fore] Same as ":cbefore", except the location list for the current window is used instead of the quickfix list.

:caf :cafter
:[count]caf[ter] Go to the [count] error after the current cursor position in the current buffer. If [count] is omitted, then 1 is used. If there are no errors, then an error message is displayed. Assumes that the entries in a quickfix list are sorted by their buffer, line and column numbers. If [count] exceeds the number of entries after the current position, then the last error in the file is selected.

:laf :lafter
:[count]laf[ter] Same as ":cafter", except the location list for the current window is used instead of the quickfix list.

:cnf :cnfile
:[count]cnf[ile][!] Display the first error in the [count] next file in

the list that includes a file name. If there are no file names at all or if there is no next file, go to the [count] next error. See :cc for [!] and 'switchbuf'.

]_CTRL-Q

]_CTRL-Q Mapped to :cnfile. default-mappings

:lnf :lnfile

:[count]lnf[ile][!] Same as ":cnfile", except the location list for the current window is used instead of the quickfix list.

]_CTRL-L

]_CTRL-L Mapped to :lnfile. default-mappings

:cpf :cpfile :cNf :cNfile

:[count]cNf[ile][!]
:[count]cpf[ile][!] Display the last error in the [count] previous file in the list that includes a file name. If there are no file names at all or if there is no next file, go to the [count] previous error. See :cc for [!] and 'switchbuf'.

[_CTRL-Q

[_CTRL-Q Mapped to :cpfile. default-mappings

:lpf :lpfile :lNf :lNfile

:[count]lNf[ile][!]
:[count]lpf[ile][!] Same as ":cNfile" and ":cpfile", except the location list for the current window is used instead of the quickfix list.

[_CTRL-L

[_CTRL-L Mapped to :lpfile. default-mappings

:crewind :cr

:cr[ewind][!] [nr] Display error [nr]. If [nr] is omitted, the FIRST error is displayed. See :cc.

[Q

[Q Mapped to :crewind. default-mappings

:lrewind :lr

:lr[ewind][!] [nr] Same as ":crewind", except the location list for the current window is used instead of the quickfix list.

[L

[L Mapped to :lrewind. default-mappings

:cfirst :cfir

:cfir[st][!] [nr] Same as ":crewind".

:lfirst :lfir

:lfir[st][!] [nr] Same as ":lrewind".

:clast :cla

:cla[st][!] [nr] Display error [nr]. If [nr] is omitted, the LAST error is displayed. See :cc.

]Q

]Q Mapped to :clast.

:llast :lla

:lla[st][!] [nr] Same as ":clast", except the location list for the current window is used instead of the quickfix list.

]L

]L Mapped to :llast.

:cq :cquit

:cq[uit][!]
:{N}cq[uit][!]
:cq[uit][!] {N} Quit Vim with error code {N}. {N} defaults to one. Useful when Vim is called from another program: e.g., a compiler will not compile the same file again, git commit will abort the committing process, fc (built-in for shells like bash and zsh) will not execute the command, etc.
{N} can also be zero, in which case Vim exits normally.
WARNING: This abandons changes in all files, like :qall!.
The [!] does not change this. Vim returns a non-zero exit code, unless {N} is zero.

:cf :cfi :cfile

:cf[ile][!] [errorfile] Read the error file and jump to the first error. This is done automatically when Vim is started with the -q option. You can use this command when you keep Vim running while compiling. If you give the name of the errorfile, the 'errorfile' option will be set to [errorfile]. See :cc for [!].
If the encoding of the error file differs from the 'encoding' option, you can use the 'makeencoding' option to specify the encoding.

:lf :lfi :lfile
:lf[ile][!] [errorfile] Same as ":cfile", except the location list for the
current window is used instead of the quickfix list.
You can not use the -q command-line option to set
the location list.

:cg[etfile] [errorfile] :cg :cgetfile
Read the error file. Just like ":cfile" but don't
jump to the first error.
If the encoding of the error file differs from the
'encoding' option, you can use the 'makeencoding'
option to specify the encoding.

:lg[etfile] [errorfile] :lg :lge :lgetfile
Same as ":cgetfile", except the location list for the
current window is used instead of the quickfix list.

:caddf :caddfile
:caddf[ile] [errorfile] Read the error file and add the errors from the
errorfile to the current quickfix list. If a quickfix
list is not present, then a new list is created.
If the encoding of the error file differs from the
'encoding' option, you can use the 'makeencoding'
option to specify the encoding.

:laddf :laddfile
:laddf[ile] [errorfile] Same as ":caddfile", except the location list for the
current window is used instead of the quickfix list.

The following commands read error messages from a buffer:
:cbuffer creates a list and jumps to its first error, :cgetbuffer creates
a list without jumping, and :caddbuffer adds entries to the current list
without changing the cursor position. The corresponding commands with an "l"
prefix use the location list.

:cb :cbuffer E681
:[range]cb[uffer][!] [bufnr]
Read the error list from the current buffer.
When [bufnr] is given it must be the number of a
loaded buffer. That buffer will then be used instead
of the current buffer.
A range can be specified for the lines to be used, for
example :10,20cbuffer. Otherwise all lines in the
buffer are used.
See :cc for [!].

:lb :lbuffer
:[range]lb[uffer][!] [bufnr]
Same as ":cbuffer", except the location list for the
current window is used instead of the quickfix list.

:cgetb :cgetbuffer
:[range]cgetb[uffer] [bufnr]
Read the error list from the current buffer. Just
like ":cbuffer" but don't jump to the first error.

:lgetb :lgetbuffer
:[range]lgetb[uffer] [bufnr]
Same as ":cgetbuffer", except the location list for
the current window is used instead of the quickfix
list.

:cad :cadd :caddbuffer
:[range]cad[dbuffer] [bufnr]
Read the error list from the current buffer and add
the errors to the current quickfix list. If a
quickfix list is not present, then a new list is
created. Otherwise, same as ":cbuffer".

:laddb :laddbuffer
:[range]laddb[uffer] [bufnr]
Same as ":caddbuffer", except the location list for
the current window is used instead of the quickfix
list.

The following commands create or update a quickfix list from an expression:
:cexpr creates a list and jumps to its first error, :cgetexpr creates a
list without jumping, and :caddexpr adds entries to the current list without
changing the cursor position. The corresponding commands with an "l" prefix
use the location list.

:cex :cexpr E777
:cex[pr][!] {expr} Create a quickfix list using the result of {expr} and
jump to the first error.
If {expr} is a String, then each newline terminated
line in the String is processed using the global value
of 'errorformat' and the result is added to the
quickfix list.
If {expr} is a List, then each String item in the list
is processed and added to the quickfix list. Non
String items in the List are ignored.
See :cc for [!].
Examples:

```
:cexpr system('grep -n xyz *')
:cexpr getline(1, '$')
```

:lex :lexpr
:lex[pr][!] {expr} Same as :cexpr, except the location list for the current window is used instead of the quickfix list.

:cgete :cgetexpr
:cgete[xpr] {expr} Create a quickfix list using the result of {expr}. Just like :cexpr, but don't jump to the first error.

:lgete :lgetexpr
:lgete[xpr] {expr} Same as :cgetexpr, except the location list for the current window is used instead of the quickfix list.

:cadde :caddexpr
:cadde[xpr] {expr} Evaluate {expr} and add the resulting lines to the current quickfix list. If a quickfix list is not present, then a new list is created. The current cursor position will not be changed. See :cexpr for more information.
Example:

```
:g/mypattern/caddexpr expand("%") .. ":" .. line(".") ..  ":" .. getline(".")
```

:lad :ladd :laddexpr
:lad[dexpr] {expr} Same as ":caddexpr", except the location list for the current window is used instead of the quickfix list.

:cl :clist
:cl[ist] [from] [, [to]]
List all errors that are valid quickfix-valid. If numbers [from] and/or [to] are given, the respective range of errors is listed. A negative number counts from the last error backwards, -1 being the last error.
The :filter command can be used to display only the quickfix entries matching a supplied pattern. The pattern is matched against the filename, module name, pattern and text of the entry.
For example, :filter /\.c:/clist lists only entries whose displayed text contains .c:. See :filter.

:cl[ist] +{count} List the current and next {count} valid errors. This is similar to ":clist from from+count", where "from" is the current error position.

:cl[ist]! [from] [, [to]]
List all errors.

:cl[ist]! +{count} List the current and next {count} error lines. This is useful to see unrecognized lines after the current one. For example, if ":clist" shows:

8384 testje.java:252: error: cannot find symbol
Then using ":cl! +3" shows the reason:

8384 testje.java:252: error: cannot find symbol
8385: ZexitCode = Fmainx();
8386: ^
8387: symbol: method Fmainx()

:lli[st] [from] [, [to]] :lli :llist
Same as ":clist", except the location list for the current window is used instead of the quickfix list.

:lli[st]! [from] [, [to]]
List all the entries in the location list for the current window.

If you insert or delete lines, mostly the correct error location is still found because hidden marks are used. Sometimes, when the mark has been deleted for some reason, the message "line changed" is shown to warn you that the error location may not be correct. If you quit Vim and start again the marks are lost and the error locations may not be correct anymore.

Two autocommands are available for running commands before and after a quickfix command (':make', ':grep' and so on) is executed. See QuickFixCmdPre and QuickFixCmdPost for details.

QuickFixCmdPost-example

When 'encoding' differs from the locale, the error messages may have a different encoding from what Vim is using. To convert the messages you can use this code:

```
function QfMakeConv()
   let qflist = getqflist()
   for i in qflist
      let i.text = iconv(i.text, "cp936", "utf-8")
   endfor
   call setqflist(qflist)
endfunction

au QuickFixCmdPost make call QfMakeConv()
```

Another option is using 'makeencoding'.

quickfix-title

Every quickfix and location list has a title. By default the title is set to

the command that created the list. The getqflist() and getloclist() functions can be used to get the title of a quickfix and a location list respectively. The setqflist() and setloclist() functions can be used to modify the title of a quickfix and location list respectively. Examples:

```
call setqflist([], 'a', {'title' : 'Cmd output'})
echo getqflist({'title' : 1})
call setloclist(3, [], 'a', {'title' : 'Cmd output'})
echo getloclist(3, {'title' : 1})
```

quickfix-index

When you jump to a quickfix/location list entry using any of the quickfix commands (e.g. :cc, :cnext, :cprev, etc.), that entry becomes the currently selected entry. The index of the currently selected entry in a quickfix/location list can be obtained using the getqflist()/getloclist() functions. Examples:

```
echo getqflist({'idx' : 0}).idx
echo getqflist({'id' : qfid, 'idx' : 0}).idx
echo getloclist(2, {'idx' : 0}).idx
```

For a new quickfix list, the first entry is selected and the index is 1. Any entry in any quickfix/location list can be set as the currently selected entry using the setqflist() function. Examples:

```
call setqflist([], 'a', {'idx' : 12})
call setqflist([], 'a', {'id' : qfid, 'idx' : 7})
call setloclist(1, [], 'a', {'idx' : 7})
```

quickfix-size

You can get the number of entries (size) in a quickfix and a location list using the getqflist() and getloclist() functions respectively. Examples:

```
echo getqflist({'size' : 1})
echo getloclist(5, {'size' : 1})
```

quickfix-context

Any Vim type can be associated as a context with a quickfix or location list. The setqflist() and the setloclist() functions can be used to associate a context with a quickfix and a location list respectively. The getqflist() and the getloclist() functions can be used to retrieve the context of a quickfix and a location list respectively. This is useful for a Vim plugin dealing with multiple quickfix/location lists.
Examples:

```
let somectx = {'name' : 'Vim', 'type' : 'Editor'}
call setqflist([], 'a', {'context' : somectx})
echo getqflist({'context' : 1})

let newctx = ['red', 'green', 'blue']
call setloclist(2, [], 'a', {'id' : qfid, 'context' : newctx})
echo getloclist(2, {'id' : qfid, 'context' : 1})
```

quickfix-parse

You can parse a list of lines using 'errorformat' without creating or modifying a quickfix list using the getqflist() function. Examples:

```
echo getqflist({'lines' : ["F1:10:Line10", "F2:20:Line20"]})
echo getqflist({'lines' : systemlist('grep -Hn quickfix *')})
```

This returns a dictionary where the "items" key contains the list of quickfix entries parsed from lines. The following shows how to use a custom 'errorformat' to parse the lines without modifying the 'errorformat' option:

```
echo getqflist({'efm' : '%f#%l#%m', 'lines' : ['F1#10#Line']})
```

To execute a command for entries or files in a list, use:

	Quickfix list	Location list
Each entry	:cdo	:ldo
Each file	:cfdo	:lfdo

Use :cdo when a command should run for every valid entry, for example :cdo s/old/new/g to change each matching line. Use :cfdo when it should run once for each file, for example :cfdo update to write each affected file once. Add the "l" prefix to use the location list instead.

The commands below describe these differences in more detail.

EXECUTE A COMMAND IN ALL THE BUFFERS IN QUICKFIX OR LOCATION LIST:

:cdo

:cdo[!] {cmd} Execute {cmd} in each valid entry in the quickfix list. It works like doing this:

```
:cfirst
:{cmd}
:cnext
:{cmd}
etc.
```

When the current file can't be abandoned and the [!]
is not present, the command fails.
When going to the next entry fails execution stops.
The last buffer (or where an error occurred) becomes
the current buffer.
{cmd} can contain '|' to concatenate several commands.

Only valid entries in the quickfix list are used.
A range can be used to select entries, e.g.:

```
:10,$cdo cmd
```

To skip entries 1 to 9.

Note: While this command is executing, the Syntax
autocommand event is disabled by adding it to
'eventignore'. This considerably speeds up editing
each buffer.
Also see :bufdo, :tabdo, :argdo, :windo,
:ldo, :cfdo and :lfdo.

:cfd :cfdo

:cfd[o][!] {cmd} Execute {cmd} in each file in the quickfix list.
It works like doing this:

```
:cfirst
:{cmd}
:cnfile
:{cmd}
etc.
```

Otherwise it works the same as :cdo.

:ld :ldo

:ld[o][!] {cmd} Execute {cmd} in each valid entry in the location list
for the current window.
It works like doing this:

```
:lfirst
:{cmd}
:lnext
:{cmd}
etc.
```

Only valid entries in the location list are used.
Otherwise it works the same as :cdo.

:lfd :lfdo

:lfd[o][!] {cmd} Execute {cmd} in each file in the location list for
the current window.
It works like doing this:

```
:lfirst
:{cmd}
:lnfile
:{cmd}
etc.
```

Otherwise it works the same as :ldo.

FILTERING A QUICKFIX OR LOCATION LIST:
cfilter-plugin :Cfilter :Lfilter package-cfilter
If you have too many entries in a quickfix list, you can use the cfilter
plugin to reduce the number of entries. Load the plugin with:

```
packadd cfilter
```

Then you can use the following commands to filter a quickfix/location list:

```
:Cfilter[!] /{pat}/
:Lfilter[!] /{pat}/
```

The :Cfilter command creates a new quickfix list from the entries matching
{pat} in the current quickfix list. {pat} is a Vim regular-expression
pattern. Both the file name and the text of the entries are matched against
{pat}. If the optional ! is supplied, then the entries not matching {pat} are
used. The pattern can be optionally enclosed using one of the following
characters: ', ", /. If the pattern is empty, then the last used search
pattern is used.

The :Lfilter command does the same as :Cfilter but operates on the current
location list.

The current quickfix/location list is not modified by these commands, so you
can go back to the unfiltered list using the :colder/|:lolder| command.

2. The error window

quickfix-window

:cope :copen w:quickfix_title

:cope[n] [height] Open a window to show the current list of errors.

When [height] is given, the window becomes that high (if there is room). When [height] is omitted the window is made ten lines high.

If there already is a quickfix window, it will be made the current window. It is not possible to open a second quickfix window. If [height] is given the existing window will be resized to it.

quickfix-buffer

The window will contain a special buffer, with 'buftype' equal to "quickfix". Don't change this! The window will have the w:quickfix_title variable set which will indicate the command that produced the quickfix list. This can be used to compose a custom status line if the value of 'statusline' is adjusted properly. Whenever this buffer is modified by a quickfix command or function, the b:changedtick variable is incremented. You can get the number of this buffer using the getqflist() and getloclist() functions by passing the "qfbufnr" item. For a location list, this buffer is wiped out when the location list is removed.

:lop :lopen

:lop[en] [height] Open a window to show the location list for the current window. Works only when the location list for the current window is present. You can have more than one location window opened at a time. Otherwise, it acts the same as ":copen".

:ccl :cclose

:ccl[ose] Close the quickfix window.

:lcl :lclose

:lcl[ose] Close the window showing the location list for the current window.

:cw :cwindow

:cw[indow] [height] Open the quickfix window when there are recognized errors. If the window is already open and there are no recognized errors, close the window.

When opening the window and [height] is given, the window becomes that high (if there is room). When [height] is omitted the window is made ten lines high.

:lw :lwindow

:lw[indow] [height] Same as ":cwindow", except use the window showing the location list for the current window.

:cbo :cbottom

:cbo[ttom] Put the cursor in the last line of the quickfix window and scroll to make it visible. This is useful for when errors are added by an asynchronous callback. Only call it once in a while if there are many updates to avoid a lot of redrawing.

:lbo :lbottom

:lbo[ttom] Same as ":cbottom", except use the window showing the location list for the current window.

Normally the quickfix window is at the bottom of the screen.
You can move the window around with window-moving commands.
For example, to move it to the top: CTRL-W K
The 'winfixheight' option will be set, which means that the window will mostly keep its height, ignoring 'winheight' and 'equalalways'. You can change the height manually (e.g., by dragging the status line above it with the mouse).

In the quickfix window, each line is one error. The line number is equal to the error number. The current entry is highlighted with the QuickFixLine highlighting. You can change it to your liking, e.g.:

```
:hi QuickFixLine ctermbg=Yellow guibg=Yellow
```

You can use ":.cc" to jump to the error under the cursor.
Hitting the <Enter> key or double-clicking the mouse on a line has the same effect. The file containing the error is opened in the window above the quickfix window. If there already is a window for that file, it is used instead. If the buffer in the used window has changed, and the error is in another file, jumping to the error will fail. You will first have to make sure the window contains a buffer which can be abandoned.

When you select a file from the quickfix window, Vim first reuses a window that already displays the file. Otherwise it uses a suitable window in the current tab page, or opens a new one. The detailed selection rules are:

1. If a window displaying the selected file is present in the current tabpage

(starting with the window before the quickfix window), then that window is used.
2. If the above step fails and if 'switchbuf' contains "usetab" and a window displaying the selected file is present in any one of the tabpages (starting with the first tabpage) then that window is used.
3. If the above step fails then a window in the current tabpage displaying a buffer with 'buftype' not set (starting with the window before the quickfix window) is used.
4. If the above step fails and if 'switchbuf' contains "uselast", then the previously accessed window is used.
5. If the above step fails then the window before the quickfix window is used. If there is no previous window, then the window after the quickfix window is used.
6. If the above step fails, then a new horizontally split window above the quickfix window is used.

quickfix-winfixbuf

When 'winfixbuf' is set for the window that would be used to display the error, jumping to an error in another buffer is not possible in that window. For a quickfix list Vim then looks for another suitable window, using the previously accessed window if it does not have 'winfixbuf' set, and otherwise splits a new window. For a location list this fails with E1513, because a location list is tied to its window and cannot move to or split off another one. Using [!] skips these checks.

CTRL-W <Enter> CTRL-W <CR>

You can use CTRL-W <Enter> to open a new window and jump to the error there.

When the quickfix window has been filled, two autocommand events are triggered. First the 'filetype' option is set to "qf", which triggers the FileType event (also see qf.vim). Then the BufReadPost event is triggered, using "quickfix" for the buffer name. This can be used to perform some action on the listed errors. Example:

```
au BufReadPost quickfix  setlocal modifiable
        \ | silent exe 'g/^/s//\=line(".") .. " "/'
        \ | setlocal nomodifiable
```

This prepends the line number to each line. Note the use of "\=" in the substitute string of the ":s" command, which is used to evaluate an expression.
The BufWinEnter event is also triggered, again using "quickfix" for the buffer name.

Note: When adding to an existing quickfix list the autocommand are not triggered.

Note: Making changes in the quickfix window has no effect on the list of errors. 'modifiable' is off to avoid making changes. If you delete or insert lines anyway, the relation between the text and the error number is messed up. If you really want to do this, you could write the contents of the quickfix window to a file and use ":cfile" to have it parsed and used as the new error list.

location-list-window

The location list window displays the entries in a location list. When you open a location list window, it is created below the current window and displays the location list for the current window. The location list window is similar to the quickfix window, except that you can have more than one location list window open at a time. When you use a location list command in this window, the displayed location list is used.

When you select a file from the location list window, Vim first uses the window associated with that location list, when possible. Otherwise it reuses a suitable window or opens a new one. The detailed selection rules are:

1. If a non-quickfix window associated with the location list is present in the current tabpage, then that window is used.
2. If the above step fails and if the file is already opened in another window in the current tabpage, then that window is used.
3. If the above step fails and 'switchbuf' contains "usetab" and if the file is opened in a window in any one of the tabpages, then that window is used.
4. If the above step fails then a window in the current tabpage showing a

buffer with 'buftype' not set is used.
5. If the above step fails, then the file is edited in a new window.

In all of the above cases, if the location list for the selected window is not yet set, then it is set to the location list displayed in the location list window.

quickfix-window-ID

You can use the getqflist() and getloclist() functions to obtain the window ID of the quickfix window and location list window respectively (if present). Examples:

```
echo getqflist({'winid' : 1}).winid
echo getloclist(2, {'winid' : 1}).winid
```

getqflist-examples

The getqflist() and getloclist() functions can be used to get the various attributes of a quickfix and location list respectively. Some examples for using these functions are below:

```
" get the title of the current quickfix list
:echo getqflist({'title' : 0}).title

" get the identifier of the current quickfix list
:let qfid = getqflist({'id' : 0}).id

" get the identifier of the fourth quickfix list in the stack
:let qfid = getqflist({'nr' : 4, 'id' : 0}).id

" check whether a quickfix list with a specific identifier exists
:if getqflist({'id' : qfid}).id == qfid

" get the index of the current quickfix list in the stack
:let qfnum = getqflist({'nr' : 0}).nr

" get the items of a quickfix list specified by an identifier
:echo getqflist({'id' : qfid, 'items' : 0}).items

" get the number of entries in a quickfix list specified by an id
:echo getqflist({'id' : qfid, 'size' : 0}).size

" get the context of the third quickfix list in the stack
:echo getqflist({'nr' : 3, 'context' : 0}).context

" get the number of quickfix lists in the stack
:echo getqflist({'nr' : '$'}).nr

" get the number of times the current quickfix list is changed
:echo getqflist({'changedtick' : 0}).changedtick

" get the current entry in a quickfix list specified by an identifier
:echo getqflist({'id' : qfid, 'idx' : 0}).idx

" get all the quickfix list attributes using an identifier
:echo getqflist({'id' : qfid, 'all' : 0})

" parse text from a List of lines and return a quickfix list
:let myList = ["a.java:10:L10", "b.java:20:L20"]
:echo getqflist({'lines' : myList}).items

" parse text using a custom 'efm' and return a quickfix list
:echo getqflist({'lines' : ['a.c#10#Line 10'], 'efm':'%f#%l#%m'}).items

" get the quickfix list window id
:echo getqflist({'winid' : 0}).winid

" get the quickfix list window buffer number
:echo getqflist({'qfbufnr' : 0}).qfbufnr

" get the context of the current location list
:echo getloclist(0, {'context' : 0}).context

" get the location list window id of the third window
:echo getloclist(3, {'winid' : 0}).winid

" get the location list window buffer number of the third window
:echo getloclist(3, {'qfbufnr' : 0}).qfbufnr

" get the file window id of a location list window (winnr: 4)
:echo getloclist(4, {'filewinid' : 0}).filewinid
```

setqflist-examples

The setqflist() and setloclist() functions can be used to set the various attributes of a quickfix and location list respectively. Some examples for using these functions are below:

```
" create an empty quickfix list with a title and a context
:let t = 'Search results'
:let c = {'cmd' : 'grep'}
:call setqflist([], ' ', {'title' : t, 'context' : c})

" set the title of the current quickfix list
:call setqflist([], 'a', {'title' : 'Mytitle'})

" change the current entry in the list specified by an identifier
:call setqflist([], 'a', {'id' : qfid, 'idx' : 10})

" set the context of a quickfix list specified by an identifier
:call setqflist([], 'a', {'id' : qfid, 'context' : {'val' : 100}})

" create a new quickfix list from a command output
:call setqflist([], ' ', {'lines' : systemlist('grep -Hn main *.c')})

" parse text using a custom efm and add to a particular quickfix list
:call setqflist([], 'a', {'id' : qfid,
            \ 'lines' : ["a.c#10#L10", "b.c#20#L20"], 'efm':'%f#%l#%m'})

" add items to the quickfix list specified by an identifier
:let newItems = [{'filename' : 'a.txt', 'lnum' : 10, 'text' : "Apple"},
                \ {'filename' : 'b.txt', 'lnum' : 20, 'text' : "Orange"}]
:call setqflist([], 'a', {'id' : qfid, 'items' : newItems})

" empty a quickfix list specified by an identifier
:call setqflist([], 'r', {'id' : qfid, 'items' : []})

" free all the quickfix lists in the stack
:call setqflist([], 'f')

" set the title of the fourth quickfix list
:call setqflist([], 'a', {'nr' : 4, 'title' : 'SomeTitle'})

" create a new quickfix list at the end of the stack
:call setqflist([], ' ', {'nr' : '$',
                    \ 'lines' : systemlist('grep -Hn class *.java')})

" create a new location list from a command output
:call setloclist(0, [], ' ', {'lines' : systemlist('grep -Hn main *.c')})

" replace the location list entries for the third window
:call setloclist(3, [], 'r', {'items' : newItems})
```

3. Using more than one list of errors quickfix-error-lists

So far it has been assumed that there is only one list of errors. Actually there can be multiple used lists that are remembered; see 'chistory' and 'lhistory'.
When starting a new list, the previous ones are automatically kept. Two commands can be used to access older error lists. They set one of the existing error lists as the current one.

:colder :col E380
:col[der] [count] Go to older error list. When [count] is given, do this [count] times. When already at the oldest error list, an error message is given.

:lolder :lol
:lol[der] [count] Same as :colder, except use the location list for the current window instead of the quickfix list.

:cnewer :cnew E381
:cnew[er] [count] Go to newer error list. When [count] is given, do this [count] times. When already at the newest error list, an error message is given.

:lnewer :lnew
:lnew[er] [count] Same as :cnewer, except use the location list for the current window instead of the quickfix list.

:chistory :chi
:[count]chi[story] Show the list of error lists. The current list is marked with ">". The output looks like:

```
  error list 1 of 3; 43 errors   :make
> error list 2 of 3; 0 errors    :helpgrep tag
  error list 3 of 3; 15 errors   :grep ex_help *.c
```

When [count] is given, then the count'th quickfix list is made the current list. Example:

```
" Make the 4th quickfix list current
:4chistory
```

:lhistory :lhi
:[count]lhi[story] Show the list of location lists, otherwise like :chistory.

When adding a new error list, it becomes the current list.

When ":colder" has been used and ":make" or ":grep" is used to add a new error list, one newer list is overwritten. This is especially useful if you are browsing with ":grep" grep. If you want to keep the more recent error lists, use ":cnewer 99" first.

To get the number of lists in the quickfix and location list stack, you can use the getqflist() and getloclist() functions respectively with the list number set to the special value '$'. Examples:

```
echo getqflist({'nr' : '$'}).nr
echo getloclist(3, {'nr' : '$'}).nr
```

To get the number of the current list in the stack:

```
echo getqflist({'nr' : 0}).nr
```

4. Using :make

:make_makeprg

:mak :make

:mak[e][!] [arguments] 1. All relevant QuickFixCmdPre autocommands are executed.

1. If the 'autowrite' option is on, write any changed buffers
2. An errorfile name is made from 'makeef'. If 'makeef' doesn't contain "##", and a file with this name already exists, it is deleted.
3. The program given with the 'makeprg' option is started (default "make") with the optional [arguments] and the output is saved in the errorfile (for Unix it is also echoed on the screen).
4. The errorfile is read using 'errorformat'.
5. All relevant QuickFixCmdPost autocommands are executed. See example below.
6. If [!] is not given the first error is jumped to.
7. The errorfile is deleted.
8. You can now move through the errors with commands like :cnext and :cprevious, see above.

This command does not accept a comment, any " characters are considered part of the arguments. If the encoding of the program output differs from the 'encoding' option, you can use the 'makeencoding' option to specify the encoding.

:lmak :lmake

:lmak[e][!] [arguments]
Same as ":make", except the location list for the current window is used instead of the quickfix list.

In short, :make runs 'makeprg', reads its output using 'errorformat', optionally jumps to the first recognized error, and then removes the temporary error file used by the normal :make workflow. If 'shellpipe' is empty, or if the program manages its own output file, this cleanup does not apply to that program's file. Use :cnext and related commands to move through the resulting list.

The ":make" command executes the command given with the 'makeprg' option. This is done by passing the command to the shell given with the 'shell' option. This works almost like typing

":!{makeprg} [arguments] {shellpipe} {errorfile}".

{makeprg} is the string given with the 'makeprg' option. Any command can be used, not just "make". Characters '%' and '#' are expanded as usual on a command-line. You can use "%<" to insert the current file name without extension, or "#<" to insert the alternate file name without extension, for example:

```
:set makeprg=make\ #<.o
```

[arguments] is anything that is typed after ":make".
{shellpipe} is the 'shellpipe' option.
{errorfile} is the 'makeef' option, with ## replaced to make it unique.

The placeholder "$*" can be used for the argument list in {makeprg} if the command needs some additional characters after its arguments. The $* is replaced then by all arguments. Example:

```
:set makeprg=latex\ \\\\nonstopmode\ \\\\input\\{$*}
```

or simpler

```
:let &mp = 'latex \\nonstopmode \\input\{$*}'
```

"$*" can be given multiple times, for example:

```
:set makeprg=gcc\ -o\ $*\ $*
```

The 'shellpipe' option defaults to "2>&1| tee" for Win32.
This means that the output of the compiler is saved in a file and not shown on the screen directly. For Unix "| tee" is used. The compiler output is shown on the screen and saved in a file the same time. Depending on the shell used "|& tee" or "2>&1| tee" is the default, so stderr output will be included.

If 'shellpipe' is empty, the {errorfile} part will be omitted. This is useful for compilers that write to an errorfile themselves.

Using QuickFixCmdPost to fix the encoding

It may be that 'encoding' is set to an encoding that differs from the messages your build program produces. This example shows how to fix this after Vim has read the error messages:

```
function QfMakeConv()
   let qflist = getqflist()
   for i in qflist
      let i.text = iconv(i.text, "cp936", "utf-8")
   endfor
   call setqflist(qflist)
endfunction

au QuickFixCmdPost make call QfMakeConv()
```

(Example by Faque Cheng)
Another option is using 'makeencoding'.

5. Using :vimgrep and :grep

grep lid

Vim has two ways to find matches for a pattern: internal and external. The advantage of the internal grep is that it works on all systems and uses the powerful Vim search patterns. An external grep program can be used when the Vim grep does not do what you want.

The internal method will be slower, because files are read into memory. For large projects, consider using :grep with an external search program. The advantages of the internal method are:

- Line separators and encoding are automatically recognized, as if a file is being edited.
- Uses Vim search patterns. Multi-line patterns can be used.
- When plugins are enabled: compressed and remote files can be searched.

gzip netrw

To be able to do this Vim loads each file as if it is being edited. When there is no match in the file the associated buffer is wiped out again. The 'hidden' option is ignored here to avoid running out of memory or file descriptors when searching many files. However, when the :hide command modifier is used the buffers are kept loaded. This makes following searches in the same files a lot faster.

Note that :copen (or :lopen for :lgrep) may be used to open a buffer containing the search results in linked form. The :silent command may be used to suppress the default full screen grep output. The ":grep!" form of the :grep command doesn't jump to the first match automatically. These commands can be combined to create a NewGrep command:

```
command! -nargs=+ NewGrep execute 'silent grep! <args>' | copen 42
```

5.1 Using Vim's internal grep

:vim :vimgrep E682 E683

:vim[grep][!] /{pattern}/[g][j][f] {file} ...
Search for {pattern} in the files {file} ... and set the error list to the matches. Files matching 'wildignore' are ignored; files in 'suffixes' are searched last.

{pattern} is a Vim search pattern. Instead of enclosing it in / any non-ID character (see 'isident') can be used, so long as it does not appear in {pattern}.
'ignorecase' applies. To overrule it put /\c in the pattern to ignore case or /\C to match case.
'smartcase' is not used.
If {pattern} is empty (e.g. // is specified), the last used search pattern is used. last-pattern

Flags:
'g' Without the 'g' flag each line is added only once. With 'g' every match is added.

'j' Without the 'j' flag Vim jumps to the first match. With 'j' only the quickfix list is updated. With the [!] any changes in the current buffer are abandoned.

'f' When the 'f' flag is specified, fuzzy string matching is used to find matching lines. The characters in {pattern} must appear in order, but need not be next to each other. In this case, {pattern} is treated as a literal string instead of a regular expression. See fuzzy-matching for more information about fuzzy matching strings.

QuickFixCmdPre and QuickFixCmdPost are triggered.
A file that is opened for matching may use a buffer number, but it is reused if possible to avoid consuming buffer numbers.

:{count}vim[grep] ...
When a number is put before the command this is used as the maximum number of matches to find. Use ":1vimgrep pattern file" to find only the first.
Useful if you only want to check if there is a match and quit quickly when it's found.

Every second or so the searched file name is displayed to give you an idea of the progress made.
Examples:

```
:vimgrep /an error/ *.c
:vimgrep /\<FileName\>/ *.h include/*
:vimgrep /myfunc/ **/*.c
```

For the use of "**" see starstar-wildcard.

:vim[grep][!] {pattern} {file} ...
Like above, but instead of enclosing the pattern in a non-ID character use a white space separated pattern. The pattern must start with an ID character.
Example:

```
:vimgrep Error *.c
```

:lv :lvimgrep
:lv[imgrep][!] /{pattern}/[g][j][f] {file} ...
:lv[imgrep][!] {pattern} {file} ...
Same as ":vimgrep", except the location list for the current window is used instead of the quickfix list.

:vimgrepa :vimgrepadd
:vimgrepa[dd][!] /{pattern}/[g][j][f] {file} ...
:vimgrepa[dd][!] {pattern} {file} ...
Just like ":vimgrep", but instead of making a new list of errors the matches are appended to the current list.

:lvimgrepa :lvimgrepadd
:lvimgrepa[dd][!] /{pattern}/[g][j][f] {file} ...
:lvimgrepa[dd][!] {pattern} {file} ...
Same as ":vimgrepadd", except the location list for the current window is used instead of the quickfix list.

5.2 External grep

Vim can interface with "grep" and grep-like programs (such as the GNU id-utils) in a similar way to its compiler integration (see :make above).

[Unix trivia: The name for the Unix "grep" command comes from ":g/re/p", where "re" stands for Regular Expression.]

:gr :grep
:gr[ep][!] [arguments] Just like ":make", but use 'grepprg' instead of 'makeprg' and 'grepformat' instead of 'errorformat'.
When 'grepprg' is "internal" this works like :vimgrep. Note that the pattern needs to be enclosed in separator characters then.
If the encoding of the program output differs from the 'encoding' option, you can use the 'makeencoding' option to specify the encoding.

:lgr :lgrep
:lgr[ep][!] [arguments] Same as ":grep", except the location list for the

current window is used instead of the quickfix list.

:grepa :grepadd

:grepa[dd][!] [arguments]
Just like ":grep", but instead of making a new list of errors the matches are appended to the current list. Example:

```
:call setqflist([])
:bufdo grepadd! something %
```

The first command makes a new error list which is empty. The second command executes "grepadd" for each listed buffer. Note the use of ! to avoid that ":grepadd" jumps to the first error, which is not allowed with :bufdo.
An example that uses the argument list and avoids errors for files without matches:

```
:silent argdo try
  \ | grepadd! something %
  \ | catch /E480:/
  \ | endtry"
```

If the encoding of the program output differs from the 'encoding' option, you can use the 'makeencoding' option to specify the encoding.

:lgrepa :lgrepadd

:lgrepa[dd][!] [arguments]
Same as ":grepadd", except the location list for the current window is used instead of the quickfix list.

5.3 Setting up external grep

If you have a standard "grep" program installed, the :grep command may work well with the defaults. The syntax is very similar to the standard command:

```
:grep foo *.c
```

Will search all files with the .c extension for the substring "foo". The arguments to :grep are passed straight to the "grep" program, so you can use whatever options your "grep" supports.

By default, :grep invokes grep with the -n option (show file and line numbers). You can change this with the 'grepprg' option. You will need to set 'grepprg' if:

a) You are using a program that isn't called "grep"
b) You have to call grep with a full path
c) You want to pass other options automatically (e.g. case insensitive search.)

Once "grep" has executed, Vim parses the results using the 'grepformat' option. This option works in the same way as the 'errorformat' option - see that for details. You may need to change 'grepformat' from the default if your grep outputs in a non-standard format, or you are using some other program with a special format.

Once the results are parsed, Vim loads the first file containing a match and jumps to the appropriate line, in the same way that it jumps to a compiler error in quickfix mode. You can then use the :cnext, :clist, etc. commands to see the other matches.

5.4 Using :grep with id-utils

You can set up :grep to work with the GNU id-utils like this:

```
:set grepprg=lid\ -Rgrep\ -s
:set grepformat=%f:%l:%m
```

then

```
:grep (regexp)
```

works just as you'd expect.
(provided you remembered to mkid first :)

5.5 Browsing source code with :vimgrep or :grep

Using the stack of error lists that Vim keeps, you can browse your files to look for functions and the functions they call. For example, suppose that you have to add an argument to the read_file() function. You enter this command:

```
:vimgrep /\<read_file\>/ *.c
```

You use ":cn" to go along the list of matches and add the argument. At one place you have to get the new argument from a higher level function msg(), and need to change that one too. Thus you use:

```
:vimgrep /\<msg\>/ *.c
```

While changing the msg() functions, you find another function that needs to
get the argument from a higher level. You can again use ":vimgrep" to find
these functions. Once you are finished with one function, you can use

```
:colder
```

to go back to the previous one.

This works like browsing a tree: ":vimgrep" goes one level deeper, creating a
list of branches. ":colder" goes back to the previous level. You can mix
this use of ":vimgrep" and "colder" to browse all the locations in a tree-like
way. If you do this consistently, you will find all locations without the
need to write down a "todo" list.

6. Selecting a compiler compiler-select

:comp :compiler E666

:comp[iler][!] {name} Set options to work with compiler {name}.
Without the "!" options are set for the
current buffer. With "!" global options are
set.
If you use ":compiler foo" in "file.foo" and
then ":compiler! bar" in another buffer, Vim
will keep on using "foo" in "file.foo".

The Vim plugins in the "compiler" directory will set options to use the
selected compiler. For :compiler local options are set, for :compiler!
global options.

In normal use, select a compiler with :compiler {name} and then run :make.
The compiler plugin usually sets 'makeprg' and 'errorformat' so Vim can run
the compiler and read its messages. The descriptions below document options
for particular compiler plugins.

The values set by a compiler plugin can be changed afterwards with :set or
:setlocal. :compiler {name} sets options for the current buffer, while
:compiler! {name} sets global options; a buffer-local value takes precedence
over a global value.

current_compiler

To support older Vim versions, the plugins always use "current_compiler" and
not "b:current_compiler". What the command actually does is the following:

- Delete the "current_compiler" and "b:current_compiler" variables.
- Define the "CompilerSet" user command. With "!" it does ":set", without "!"
  it does ":setlocal".
- Execute ":runtime! compiler/{name}.{vim,lua}". The plugins are expected to
  set options with "CompilerSet" and set the "current_compiler" variable to the
  name of the compiler.
- Delete the "CompilerSet" user command.
- Set "b:current_compiler" to the value of "current_compiler".
- Without "!" the old value of "current_compiler" is restored.

For writing a compiler plugin, see write-compiler-plugin.

Use the compiler-make plugin to undo the effect of a compiler plugin.

BIOME compiler-biome quickfix-biome

Biome check lints JavaScript, TypeScript, JSX, TSX, JSON, JSONC, HTML, Vue,
Svelte, Astro, CSS, GraphQL and GritQL files.

Commonly used compiler options can be added to 'makeprg' by setting the
b/g:biome_makeprg_params variable. For example (global default is ""):

```
let b:biome_makeprg_params = "--diagnostic-level=error --staged"
```

CPPCHECK quickfix-cppcheck compiler-cppcheck

Use g/b:`c_cppcheck_params` to set cppcheck parameters. The global
settings by default include

- --verbose: Enables verbose output.
- --force: Forces checking of all configurations.

- --inline-suppr: Allows inline suppressions.

- --enable=...: Enables specific checks like warnings, style, performance,

  portability, information, and missing includes.
- -j: Utilizes multiple processors if available, determined by the

  getconf command if available (requires omitting the unusedFunction check)

For C++ files (filetype == 'cpp'), the --language=c++ option is added to ensure Cppcheck treats the file as C++.

If compile_commands.json is present in the current directory, it is added as a --project parameter to the command line. Otherwise, by default the directories in &path are passed as include directories. These can be set by g/b:`c_cppcheck_includes` as a list of -I flags. Tim Pope's vim-apathy plug-in [0] can expand &path. To also append the folders in a git repo use

```
let &l:path = join(systemlist('git ls-tree -d --name-only -r HEAD'), ',')
```

[0] https://github.com/tpope/vim-apathy

DOTNET compiler-dotnet

The .NET CLI compiler outputs both errors and warnings by default. The output may be limited to include only errors, by setting the g:dotnet_errors_only variable to v:true.

The associated project name is included in each error and warning. To suppress the project name, set the g:dotnet_show_project_file variable to v:false.

Example: limit output to only display errors, and suppress the project name:

```
let dotnet_errors_only = v:true
let dotnet_show_project_file = v:false
compiler dotnet
```

GCC quickfix-gcc compiler-gcc

There's one variable you can set for the GCC compiler:

g:compiler_gcc_ignore_unmatched_lines
Ignore lines that don't match any patterns defined for GCC. Useful if output from commands run from make are generating false positives.

JAVAC compiler-javac

Commonly used compiler options can be added to 'makeprg' by setting the b/g:javac_makeprg_params variable. For example:

```
let g:javac_makeprg_params = "-Xlint:all -encoding utf-8"
```

MAVEN compiler-maven

Commonly used compiler options can be added to 'makeprg' by setting the b/g:maven_makeprg_params variable. For example:

```
let g:maven_makeprg_params = "-DskipTests -U -X"
```

SPOTBUGS compiler-spotbugs

SpotBugs is a static analysis tool that can be used to find bugs in Java. It scans the Java bytecode of all classes in the currently open buffer. (Therefore, :compiler! spotbugs is not supported.)

Commonly used compiler options can be added to 'makeprg' by setting the "b:" or "g:spotbugs_makeprg_params" variable. For example:

```
let b:spotbugs_makeprg_params = "-longBugCodes -effort:max -low"
```

The global default is "-workHard -experimental".

By default, the class files are searched in the directory where the source files are placed. However, typical Java projects use distinct directories for source files and class files. To make both known to SpotBugs, assign their paths (distinct and relative to their common root directory) to the following properties (using the example of a common Maven project):

```
let g:spotbugs_properties = {
        \ 'sourceDirPath':      ['src/main/java'],
        \ 'classDirPath':       ['target/classes'],
        \ 'testSourceDirPath':  ['src/test/java'],
        \ 'testClassDirPath':   ['target/test-classes'],
\ }
```

Note that source and class path entries are expected to come in pairs: define
both "sourceDirPath" and "classDirPath" when you are considering at least one,
and apply the same logic to "testSourceDirPath" and "testClassDirPath".
Note that values for the path keys describe only for SpotBugs where to look
for files; refer to the documentation for particular compiler plugins for more
information.

The default pre- and post-compiler actions are provided for Ant, Maven, and
Javac compiler plugins and can be selected by assigning the name of a compiler
plugin (ant, maven, or javac) to the "compiler" key:

```
let g:spotbugs_properties = {
        \ 'compiler':           'maven',
\ }
```

This single setting is essentially equivalent to all the settings below, with
the exception made for the "PreCompilerAction" and "PreCompilerTestAction"
values: their listed Funcrefs will obtain no-op implementations whereas the
implicit Funcrefs of the "compiler" key will obtain the requested defaults if
available.

```
let g:spotbugs_properties = {
        \ 'PreCompilerAction':
                \ function('spotbugs#DefaultPreCompilerAction'),
        \ 'PreCompilerTestAction':
                \ function('spotbugs#DefaultPreCompilerTestAction'),
        \ 'PostCompilerAction':
                \ function('spotbugs#DefaultPostCompilerAction'),
        \ 'sourceDirPath':      ['src/main/java'],
        \ 'classDirPath':       ['target/classes'],
        \ 'testSourceDirPath':  ['src/test/java'],
        \ 'testClassDirPath':   ['target/test-classes'],
\ }
```

With default actions, the compiler of choice will attempt to rebuild the class
files for the buffer (and possibly for the whole project) as soon as a Java
syntax file is loaded; then, spotbugs will attempt to analyze the quality of
the compilation unit of the buffer.

Vim commands proficient in 'makeprg' [0] can be composed with default actions.
Begin by considering which of the supported keys, "DefaultPreCompilerCommand",
"DefaultPreCompilerTestCommand", or "DefaultPostCompilerCommand", you need to
write an implementation for, observing that each of these keys corresponds to
a particular "*Action" key. Follow it by defining a new function that always
declares an only parameter of type string and puts to use a command equivalent
of :make, and assigning its Funcref to the selected key. For example:

```
function! GenericPostCompilerCommand(arguments) abort
        execute 'make ' .. a:arguments
endfunction

let g:spotbugs_properties = {
        \ 'DefaultPostCompilerCommand':
                \ function('GenericPostCompilerCommand'),
\ }
```

When "PostCompilerAction" is available, "PostCompilerActionExecutor" is also
supported. Its value must be a Funcref pointing to a function that always
declares a single parameter of type string and decides whether :execute can
be dispatched on its argument, containing a pending post-compiler action,
after ascertaining the current status of :cc (or :ll):

```
function! GenericPostCompilerActionExecutor(action) abort
        try
                cc
        catch /\<E42:/
                execute a:action
        endtry
endfunction
```

Complementary, some or all of the available "Pre*Action"s (or "*Pre*Command"s)
may run :doautocmd java_spotbugs_post User in their implementations before
:make (or its equivalent) to define a once-only ShellCmdPost :autocmd

that will arrange for "PostCompilerActionExecutor" to be invoked; and then run
:doautocmd java_spotbugs_post ShellCmdPost to consume this event:

```
function! GenericPreCompilerCommand(arguments) abort
        if !exists('g:spotbugs_compilation_done')
                doautocmd java_spotbugs_post User
                execute 'make ' .. a:arguments
                " only run doautocmd when :make was synchronous
                " see note below
                doautocmd java_spotbugs_post ShellCmdPost " XXX: (a)
                let g:spotbugs_compilation_done = 1
        else
                cc
        endif
endfunction

function! GenericPreCompilerTestCommand(arguments) abort
        if !exists('g:spotbugs_test_compilation_done')
                doautocmd java_spotbugs_post User
                execute 'make ' .. a:arguments
                " only run doautocmd when :make was synchronous
                " see note below
                doautocmd java_spotbugs_post ShellCmdPost " XXX: (b)
                let g:spotbugs_test_compilation_done = 1
        else
                cc
        endif
endfunction

let g:spotbugs_properties = {
        \ 'compiler':           'maven',
        \ 'DefaultPreCompilerCommand':
                \ function('GenericPreCompilerCommand'),
        \ 'DefaultPreCompilerTestCommand':
                \ function('GenericPreCompilerTestCommand'),
        \ 'PostCompilerActionExecutor':
                \ function('GenericPostCompilerActionExecutor'),
\ }
```

If a command equivalent of :make is capable of asynchronous execution and
consuming ShellCmdPost events, :doautocmd java_spotbugs_post ShellCmdPost
must be removed from such "*Action" (or "*Command") implementations (i.e. the
lines (a) and (b) in the listed examples) to retain a sequential order for
non-blocking execution, and any notification (see below) must be suppressed.
A ShellCmdPost :autocmd can be associated with any :augroup by assigning
its name to the "augroupForPostCompilerAction" key.

When default actions are not suited to a desired workflow, proceed by writing
arbitrary functions yourself and matching their Funcrefs to the supported
keys: "PreCompilerAction", "PreCompilerTestAction", and "PostCompilerAction".

The next example re-implements the default pre-compiler actions for a Maven
project and requests other default Maven settings with the "compiler" entry:

```
function! MavenPreCompilerAction() abort
        call spotbugs#DeleteClassFiles()
        compiler maven
        make compile
        cc
endfunction

function! MavenPreCompilerTestAction() abort
        call spotbugs#DeleteClassFiles()
        compiler maven
        make test-compile
        cc
endfunction

let g:spotbugs_properties = {
        \ 'compiler':           'maven',
        \ 'PreCompilerAction':
                \ function('MavenPreCompilerAction'),
        \ 'PreCompilerTestAction':
                \ function('MavenPreCompilerTestAction'),
\ }
```

Note that all entered custom settings will take precedence over the matching
default settings in "g:spotbugs_properties".
Note that it is necessary to notify the plugin of the result of a pre-compiler
action before further work can be undertaken. Using :cc after :make (or
:ll after :lmake) as the last command of an action is the supported means
of such communication.

Two commands, "SpotBugsRemoveBufferAutocmd" and "SpotBugsDefineBufferAutocmd",
are provided to toggle actions for buffer-local autocommands. For example, to
also run actions on any BufWritePost and Signal event, add these lines to
~/.config/nvim/after/ftplugin/java.vim:

```
if exists(':SpotBugsDefineBufferAutocmd') == 2
        SpotBugsDefineBufferAutocmd BufWritePost Signal
endif
```

Otherwise, you can turn to :doautocmd java_spotbugs User at any time.

The "g:spotbugs_properties" variable is consulted by the Java filetype plugin (ft-java-plugin) to arrange for the described automation, and, therefore, it must be defined before FileType events can take place for the buffers loaded with Java source files. It could, for example, be set in a project-local vimrc loaded by [1].

Both "g:spotbugs_properties" and "b:spotbugs_properties" are recognized and must be modifiable (:unlockvar). The "*Command" entries are always treated as global functions to be shared among all Java buffers.

The SpotBugs Java library and, by extension, its distributed shell scripts do not support in the -textui mode listed pathnames with directory filenames that contain blank characters [2]. To work around this limitation, consider making a symbolic link to such a directory from a directory that does not have blank characters in its name and passing this information to SpotBugs:

```
let g:spotbugs_alternative_path = {
        \ 'fromPath':   'path/to/dir_without_blanks',
        \ 'toPath':     'path/to/dir with blanks',
\ }
```

[0] https://github.com/Konfekt/vim-compilers
[1] https://github.com/MarcWeber/vim-addon-local-vimrc
[2] https://github.com/spotbugs/spotbugs/issues/909

GNU MAKE compiler-make

Since the default make program is "make", the compiler plugin for make, :compiler make, will reset the 'makeprg' and 'errorformat' option to the default values and unlet any variables that may have been set by a previous compiler plugin.

GROFF quickfix-groff compiler-groff

The GROFF compiler plugin uses the mom macro set (documented in the groff_mom manpage) as input and expects that the output file type extension is passed to make, say :make html or :make pdf.

Additional arguments can be passed to groff by setting them in b:groff_compiler_args or g:groff_compiler_args. The language argument passed to groff is set using 'spelllang'; it can be overridden by setting b:groff_compiler_lang. The default encoding is UTF-8 and can be changed by setting b:groff_compiler_encoding or g:groff_compiler_encoding.

PANDOC quickfix-pandoc compiler-pandoc

The Pandoc compiler plugin expects that an output file type extension is passed to make, say :make html or :make pdf.

Additional arguments can be passed to pandoc:

- either by appending them to make, say :make html --self-contained .

- or setting them in b:pandoc_compiler_args or g:pandoc_compiler_args.

The --from argument is an educated guess using the buffer file type;
it can be overridden by setting b:pandoc_compiler_from.
The --metadata lang argument is set using 'spelllang';
If --from=markdown is assumed and no title set in a title header or
YAML block, then the filename (without extension) is used as the title.

PERL quickfix-perl compiler-perl

The Perl compiler plugin doesn't actually compile, but invokes Perl's internal syntax checking feature and parses the output for possible errors so you can correct them in quick-fix mode.

Warnings are forced regardless of "no warnings" or "$^W = 0" within the file being checked. To disable this set g:perl_compiler_force_warnings to a zero value. For example:

```
let g:perl_compiler_force_warnings = 0
```

MYPY TYPE CHECKER compiler-mypy

Commonly used compiler options can be added to 'makeprg' by setting the b/g:mypy_makeprg_params variable. For example:

```
let b:mypy_makeprg_params = "--warn-unused-ignores"
```

The global default is "--strict --ignore-missing-imports".

PYRIGHT TYPE CHECKER

compiler-pyright

Commonly used compiler options can be added to 'makeprg' by setting the
b/g:pyright_makeprg_params variable.

The global default is "pyright".

TY TYPE CHECKER

compiler-ty

Commonly used compiler options and executable can be set by the
b/g:ty_makeprg variable. For example:

```
let b:ty_makeprg = "uv run ty"
```

The global default is "ty --no-progress --color=never".

RUFF LINTER

compiler-ruff

Commonly used compiler options can be added to 'makeprg' by setting the
b/g:ruff_makeprg_params variable. For example:

```
let b:ruff_makeprg_params = "--max-line-length "..&textwidth
```

The global default is "--preview".

PYLINT LINTER

compiler-pylint

Commonly used compiler options can be added to 'makeprg' by setting the
b/g:pylint_makeprg_params variable. For example:

```
let b:pylint_makeprg_params = "--max-line-length "..&textwidth
```

The global default is "--jobs=0" to use (almost) all cores.

PYUNIT COMPILER

compiler-pyunit

This is not actually a compiler, but a unit testing framework for the
Python language. It is included into standard Python distribution
starting from version 2.0. For older versions, you can get it from
https://pyunit.sourceforge.net.

When you run your tests with the help of the framework, possible errors
are parsed by Vim and presented for you in quick-fix mode.

Unfortunately, there is no standard way to run the tests.
The alltests.py script seems to be used quite often, that's all.
Useful values for the 'makeprg' options therefore are:
```
 setlocal makeprg=./alltests.py " Run a testsuite
 setlocal makeprg=python\ %:S   " Run a single testcase
```

PYTEST COMPILER

compiler-pytest

Commonly used compiler options can be added to 'makeprg' by setting the
b/g:pytest_makeprg_params variable. For example:

```
let b:pytest_makeprg_params = "--verbose --no-summary --disable-warnings"
```

The global default is "--tb=short --quiet"; Python warnings are suppressed.

TEX COMPILER

compiler-tex

Included in the distribution compiler for TeX ($VIMRUNTIME/compiler/tex.vim)
uses make command if possible. If the compiler finds a file named "Makefile"
or "makefile" in the current directory, it supposes that you want to process
your *TeX files with make, and the makefile does the right work. In this case
compiler sets 'errorformat' for *TeX output and leaves 'makeprg' untouched.
If neither "Makefile" nor "makefile" is found, the compiler will not use make.
You can force the compiler to ignore makefiles by defining
b:tex_ignore_makefile or g:tex_ignore_makefile variable (they are checked for
existence only).

If the compiler chose not to use make, it needs to choose a right program for
processing your input. If b:tex_flavor or g:tex_flavor (in this precedence)
variable exists, it defines TeX flavor for :make (actually, this is the name
of executed command), and if both variables do not exist, it defaults to
"latex". For example, while editing chapter2.tex \input-ed from mypaper.tex
written in AMS-TeX:

```
:let b:tex_flavor = 'amstex'
:compiler tex
```

```
        [editing...]
```

```
:make mypaper
```

Note that you must specify a name of the file to process as an argument (to process the right file when editing \input-ed or \include-ed file; portable solution for substituting % for no arguments is welcome). This is not in the semantics of make, where you specify a target, not source, but you may specify filename without extension ".tex" and mean this as "make filename.dvi or filename.pdf or filename.some_result_extension according to compiler".

Note: tex command line syntax is set to usable both for MikTeX (suggestion by Srinath Avadhanula) and teTeX (checked by Artem Chuprina). Suggestion from errorformat-LaTeX is too complex to keep it working for different shells and OSes and also does not allow to use other available TeX options, if any. If your TeX doesn't support "-interaction=nonstopmode", please report it with different means to express \nonstopmode from the command line.

TOMBI quickfix-toml compiler-tombi

The tombi compiler plugin does not compile.

It runs "tombi lint" and parses diagnostics into the quickfix list.

Color codes are stripped from the linter output to keep errorformat parsing reliable. This may require a working "sed" for old versions of the tombi linter.

TSC COMPILER compiler-tsc

The executable and compiler options can be added to 'makeprg' by setting the b/g:tsc_makeprg variable. For example:

```
let b:tsc_makeprg = "npx tsc --noEmit"
```

TYPST COMPILER compiler-typst

Vim includes a compiler plugin for Typst files. This compiler is enabled automatically in Typst buffers by the Typst filetype plugin ft-typst-plugin. Run :make to compile the current Typst file.

g:typst_cmd

By default Vim will use "typst" as the command to run the Typst compiler. This can be changed by setting the g:typst_cmd variable:

```
let g:typst_cmd = "/path/to/other/command"
```

7. The error format error-file-format

errorformat E372 E373 E374 E375 E376 E377 E378

The 'errorformat' option specifies patterns that Vim uses to read compiler output. A pattern can extract the file name, line number, column and message from each line. The first format that matches with an error message is used. You can add several formats for different messages your compiler produces, or even entries for multiple compilers. See efm-entries.

Each entry in 'errorformat' is a scanf-like string that describes the format. First, you need to know how scanf works. Look in the documentation of your C compiler. Below you find the % items that Vim understands. Others are invalid.

Quick start: Vim already includes formats for many common compilers. Check errorformats for an example before writing your own. If no example matches, start with the basic items below and adapt one of the examples as needed.

Special characters in 'errorformat' are comma and backslash. See efm-entries for how to deal with them. Note that a literal "%" is matched by "%%", thus it is not escaped with a backslash.
Keep in mind that in the :make and :grep output all NUL characters are replaced with SOH (0x01).

Note: By default the difference between upper and lowercase is ignored. If you want to match case, add "\C" to the pattern /\C.

Vim will read lines of any length, but only the first 4095 bytes are used, the rest is ignored. Items can only be 1023 bytes long.

Basic items

%f	file name (finds a string)
%b	buffer number (finds a number)
%o	module name (finds a string)
%l	line number (finds a number)
%e	end line number (finds a number)
%c	column number (finds a number representing character column of the error, byte index, a <tab> is 1 character column)
%v	virtual column number (finds a number representing the screen column of the error, where a <tab> is always 8 screen columns regardless of 'tabstop')
%k	end column number (finds a number representing the character column of the error, byte index, or a number representing screen end column of the error if

```
                    it's used with %v)
        %t          error type (finds a single character):
                        e - error message
                        w - warning message
                        i - info message
                        n - note message
                    The uppercase letters E, W, I and N are also
                    recognized and have the same meaning.
        %n          error number (finds a number)
        %m          error message (finds a string)
        %r          matches the "rest" of a single-line file message %O/P/Q
        %p          pointer line (finds a sequence of '-', '.', ' ' or
                    tabs and uses the length for the column number)
        %*{conv}    any scanf non-assignable conversion
        %%          the single '%' character
        %s          search text (finds a string)
```

The "%f" conversion may depend on the current 'isfname' setting. "~/" is expanded to the home directory and environment variables are expanded.

The "%f" and "%m" conversions have to detect the end of the string. This normally happens by matching following characters and items. When nothing is following the rest of the line is matched. If "%f" is followed by a '%' or a backslash, it will look for a sequence of 'isfname' characters.

On Windows a leading "C:" will be included in "%f", even when using "%f:". This means that a file name which is a single alphabetical letter will not be detected.

The "%b" conversion is used to parse a buffer number. This is useful for referring to lines in a scratch buffer or a buffer with no name. If a buffer with the matching number doesn't exist, then that line is used as a non-error line.

The "%p" conversion is normally followed by a "^". It's used for compilers that output a line like:

```
^
```

or

```
---------^
```

to indicate the column of the error. This is to be used in a multi-line error message. See errorformat-javac for a useful example.

The "%s" conversion specifies the text to search for, to locate the error line. The text is used as a literal string. The anchors "^" and "$" are added to the text to locate the error line exactly matching the search text and the text is prefixed with the "\V" atom to make it "very nomagic". The "%s" conversion can be used to locate lines without a line number in the error output. Like the output of the "grep" shell command.
When the pattern is present the line number will not be used.

The "%o" conversion specifies the module name in quickfix entry. If present it will be used in quickfix error window instead of the filename. The module name is used only for displaying purposes, the file name is used when jumping to the file.

Changing directory

The following uppercase conversion characters specify the type of special format strings. At most one of them may be given as a prefix at the beginning of a single comma-separated format pattern.
Some compilers produce messages that consist of directory names that have to be prepended to each file name read by %f (example: GNU make). The following codes can be used to scan these directory names; they will be stored in an internal directory stack. E379

```
        %D          "enter directory" format string; expects a following
                      %f that finds the directory name
        %X          "leave directory" format string; expects following %f
```

When defining an "enter directory" or "leave directory" format, the "%D" or "%X" has to be given at the start of that substring. Vim tracks the directory changes and prepends the current directory to each erroneous file found with a relative path. See quickfix-directory-stack for details, tips and limitations.

Multi-line messages errorformat-multi-line

It is possible to read the output of programs that produce multi-line messages, i.e. error strings that consume more than one line. Possible prefixes are:

```
        %E          start of a multi-line error message
        %W          start of a multi-line warning message
        %I          start of a multi-line informational message
        %N          start of a multi-line note message
        %A          start of a multi-line message (unspecified type)
        %>          for next line start with current pattern again efm-%>
        %C          continuation of a multi-line message
        %Z          end of a multi-line message
```

These can be used with '+' and '-', see efm-ignore below.

Using "\n" in the pattern won't work to match multi-line messages.

Example: Your compiler happens to write out errors in the following format
(leading line numbers not being part of the actual output):

1 Error 275
2 line 42
3 column 3
4 ' ' expected after '--'

The appropriate error format string has to look like this:

```
:set efm=%EError\ %n,%Cline\ %l,%Ccolumn\ %c,%Z%m
```

And the :clist error message generated for this error is:

```
 1:42 col 3 error 275:  ' ' expected after '--'
```

Another example: Think of a Python interpreter that produces the following
error message (line numbers are not part of the actual output):

```
 1  ======================================================================
 2  FAIL: testGetTypeIdCachesResult (dbfacadeTest.DjsDBFacadeTest)
 3  ----------------------------------------------------------------------
 4  Traceback (most recent call last):
 5    File "unittests/dbfacadeTest.py", line 89, in testFoo
 6      self.assertEquals(34, dtid)
 7    File "/usr/lib/python3.8/unittest.py", line 286, in
 8   failUnlessEqual
 9      raise self.failureException, \
10  AssertionError: 34 != 33
11
12  ----------------------------------------------------------------------
13  Ran 27 tests in 0.063s
```

Say you want :clist write the relevant information of this message only,
namely:

```
 5 unittests/dbfacadeTest.py:89:  AssertionError: 34 != 33
```

Then the error format string could be defined as follows:

```
:set efm=%C\ %.%#,%A\ \ File\ \"%f\"\\,\ line\ %l%.%#,%Z%[%^\ ]%\\@=%m
```

Note that the %C string is given before the %A here: since the expression
' %.%#' (which stands for the regular expression ' .*') matches every line
starting with a space, followed by any characters to the end of the line,
it also hides line 7 which would trigger a separate error message otherwise.
Error format strings are always parsed pattern by pattern until the first
match occurs.

efm-%>

The %> item can be used to avoid trying patterns that appear earlier in
'errorformat'. This is useful for patterns that match just about anything.
For example, if the error looks like this:

```
Error in line 123 of foo.c:
unknown variable "i"
```

This can be found with:

```
:set efm=xxx,%E%>Error in line %l of %f:,%Z%m
```

Where "xxx" has a pattern that would also match the second line.

Important: There is no memory of what part of the errorformat matched before;
every line in the error file gets a complete new run through the error format
lines. For example, if one has:

```
setlocal efm=aa,bb,cc,dd,ee
```

Where aa, bb, etc. are error format strings. Each line of the error file will
be matched to the pattern aa, then bb, then cc, etc. Just because cc matched
the previous error line does _not_ mean that dd will be tried first on the
current line, even if cc and dd are multi-line errorformat strings.

Separate file name errorformat-separate-filename

These prefixes are useful if the file name is given once and multiple messages
follow that refer to this file name.

```
        %O              single-line file message: overread the matched part
        %P              single-line file message: push file %f onto the stack
        %Q              single-line file message: pop the last file from stack
```

Example: Given a compiler that produces the following error logfile (without
leading line numbers):

```
 1  [a1.tt]
 2  (1,17)  error: ';' missing
 3  (21,2)  warning: variable 'z' not defined
 4  (67,3)  error: end of file found before string ended
 5
 6  [a2.tt]
 7
 8  [a3.tt]
```

```
     9  NEW compiler v1.1
    10  (2,2)   warning: variable 'x' not defined
    11  (67,3)  warning: 's' already defined
```

This logfile lists several messages for each file enclosed in [...] which are
properly parsed by an error format like this:

```
:set efm=%+P[%f],(%l\\,%c)%*[\ ]%t%*[^:]:\ %m,%-Q
```

A call of :clist writes them accordingly with their correct filenames:

```
  2 a1.tt:1 col 17 error: ';' missing
  3 a1.tt:21 col 2 warning: variable 'z' not defined
  4 a1.tt:67 col 3 error: end of file found before string ended
  8 a3.tt:2 col 2 warning: variable 'x' not defined
  9 a3.tt:67 col 3 warning: 's' already defined
```

Unlike the other prefixes that all match against whole lines, %P, %Q and %O
can be used to match several patterns in the same line. Thus it is possible
to parse even nested files like in the following line:

```
{"file1" {"file2" error1} error2 {"file3" error3 {"file4" error4 error5}}}
```

The %O then parses over strings that do not contain any push/pop file name
information. See errorformat-LaTeX for an extended example.

Ignoring and using whole messages efm-ignore

The codes '+' or '-' can be combined with the uppercase codes above; in that
case they have to precede the letter, e.g. '%+A' or '%-G':

%-	do not include the matching multi-line in any output
%+	include the whole matching line in the %m error string

One prefix is only useful in combination with '+' or '-', namely %G. It
parses over lines containing general information like compiler version strings
or other headers that can be skipped.

%-G	ignore this message
%+G	general message

Pattern matching

The scanf()-like "%*[]" notation is supported for backward-compatibility
with previous versions of Vim. However, it is also possible to specify
(nearly) any Vim supported regular expression in format strings.
Since meta characters of the regular expression language can be part of
ordinary matching strings or file names (and therefore internally have to
be escaped), meta symbols have to be written with leading '%':

%\	The single '\' character. Note that this has to be escaped ("%\\") in ":set errorformat=" definitions.
%.	The single '.' character.
%#	The single "*"(!) character.
%^	The single '^' character. Note that this is not useful, the pattern already matches start of line.
%$	The single '$' character. Note that this is not useful, the pattern already matches end of line.
%[	The single '[' character for a [] character range.
%~	The single '~' character.

When using character classes in expressions (see /\i for an overview),
terms containing the "\+" quantifier can be written in the scanf() "%*"
notation. Example: "%\\d%\\+" ("\d\+", "any number") is equivalent to
"%*\\d".
Important note: The \(...\) grouping of sub-matches can not be used in format
specifications because it is reserved for internal conversions.

Multiple entries in 'errorformat' efm-entries

To be able to detect output from several compilers, several format patterns
may be put in 'errorformat', separated by commas (note: blanks after the comma
are ignored). The first pattern that has a complete match is used. If no
match is found, matching parts from the last one will be used, although the
file name is removed and the error message is set to the whole message. If
there is a pattern that may match output from several compilers (but not in a
right way), put it after one that is more restrictive.

To include a comma in a pattern precede it with a backslash (you have to type
two in a ":set" command). To include a backslash itself give two backslashes
(you have to type four in a ":set" command). You also need to put a backslash
before a space for ":set".

Valid matches quickfix-valid

If a line does not completely match one of the entries in 'errorformat', the
whole line is put in the error message and the entry is marked "not valid"
These lines are skipped with the ":cn" and ":cp" commands (unless there is
no valid line at all). You can use ":cl!" to display all the error messages.

If the error format does not contain a file name Vim cannot switch to the
correct file. You will have to do this by hand.

For example, the format of the output from the Amiga Aztec compiler is:

```
filename>linenumber:columnnumber:errortype:errornumber:errormessage
```

filename	name of the file in which the error was detected
linenumber	line number where the error was detected
columnnumber	column number where the error was detected
errortype	type of the error, normally a single 'E' or 'W'

```
        errornumber     number of the error (for lookup in the manual)
        errormessage    description of the error
```

This can be matched with this 'errorformat' entry:

```
        %f>%l:%c:%t:%n:%m
```

Some examples for C compilers that produce single-line error outputs:

```
%f:%l:\ %t%*[^0123456789]%n:\ %m        for Manx/Aztec C error messages
                                        (scanf() doesn't understand [0-9])
%f\ %l\ %t%*[^0-9]%n:\ %m               for SAS C
\"%f\"\\,%*[^0-9]%l:\ %m                for generic C compilers
%f:%l:\ %m                              for GCC
%f:%l:\ %m,%Dgmake[%*\\d]:\ Entering\ directory\ `%f',
%Dgmake[%*\\d]:\ Leaving\ directory\ `%f'
                                        for GCC with gmake (concat the lines!)
%f(%l)\ :\ %*[^:]:\ %m                  old SCO C compiler (pre-OS5)
%f(%l)\ :\ %t%*[^0-9]%n:\ %m            idem, with error type and number
%f:%l:\ %m,In\ file\ included\ from\ %f:%l:,\^I\^Ifrom\ %f:%l%m
                                        for GCC, with some extras
```

Extended examples for the handling of multi-line messages are given below, see errorformat-Jikes and errorformat-LaTeX.

Note the backslash in front of a space and double quote. It is required for the :set command. There are two backslashes in front of a comma, one for the :set command and one to avoid recognizing the comma as a separator of error formats.

Filtering messages

If you have a compiler that produces error messages that do not fit in the format string, you could write a program that translates the error messages into this format. You can use this program with the ":make" command by changing the 'makeprg' option. For example:

```
:set mp=make\ \\\|&\ error_filter
```

The backslashes before the pipe character are required to avoid it to be recognized as a command separator. The backslash before each space is required for the set command.

8. The directory stack quickfix-directory-stack

Quickfix maintains a stack for saving all used directories parsed from the make output. For GNU-make this is rather simple, as it always prints the absolute path of all directories it enters and leaves. Regardless if this is done via a 'cd' command in the makefile or with the parameter "-C dir" (change to directory before reading the makefile). It may be useful to use the switch "-w" to force GNU-make to print out the working directory before and after processing.

This matters only when compiler output contains relative file names from multiple directories. If the output does not clearly report directory changes, Vim has to guess and can select the wrong file. Printing absolute directory names and both entering and leaving directory messages avoids this problem.

Maintaining the correct directory is more complicated if you don't use GNU-make. AIX-make for example doesn't print any information about its working directory. Then you need to enhance the makefile. In the makefile of LessTif there is a command which echoes "Making {target} in {dir}". The special problem here is that it doesn't print information on leaving the directory and that it doesn't print the absolute path.

To solve the problem with relative paths and missing "leave directory" messages Vim uses the following algorithm:

1) Check if the given directory is a subdirectory of the current directory.
   If this is true, store it as the current directory.
2) If it is not a subdir of the current directory, try if this is a
   subdirectory of one of the upper directories.
3) If the directory still isn't found, it is assumed to be a subdirectory
   of Vim's current directory.

Additionally it is checked for every file, if it really exists in the identified directory. If not, it is searched in all other directories of the directory stack (NOT the directory subtree!). If it is still not found, it is assumed that it is in Vim's current directory.

There are limitations in this algorithm. These examples assume that make just prints information about entering a directory in the form "Making all in dir".

1) Assume you have following directories and files:
   ./dir1
   ./dir1/file1.c
   ./file1.c

   If make processes the directory "./dir1" before the current directory and there is an error in the file "./file1.c", you will end up with the file "./dir1/file.c" loaded by Vim.

   This can only be solved with a "leave directory" message.

2) Assume you have following directories and files:
   ./dir1

```
./dir1/dir2
./dir2
```

You get the following:

```
Make output                    Directory interpreted by Vim
-----------------------        ----------------------------
Making all in dir1             ./dir1
Making all in dir2             ./dir1/dir2
Making all in dir2             ./dir1/dir2
```

This can be solved by printing absolute directories in the "enter directory" message or by printing "leave directory" messages.

To avoid this problem, ensure to print absolute directory names and "leave directory" messages.

Examples for Makefiles:

Unix:

```
    libs:
            for dn in $(LIBDIRS); do                            \
                (cd $$dn; echo "Entering dir '$$(pwd)'"; make); \
                echo "Leaving dir";                             \
            done
```

Add

```
    %DEntering\ dir\ '%f',%XLeaving\ dir
```

to your 'errorformat' to handle the above output.

Note that Vim doesn't check if the directory name in a "leave directory" messages is the current directory. This is why you could just use the message "Leaving dir".

9. Specific error file formats errorformats

errorformat-Jikes

Jikes(TM), a source-to-bytecode Java compiler published by IBM Research, produces simple multi-line error messages.

An 'errorformat' string matching the produced messages is shown below. The following lines can be placed in the user's init.vim to overwrite Vim's recognized default formats, or see :set+= how to install this format additionally to the default.

```
:set efm=%A%f:%l:%c:%*\\d:%*\\d:,
      \%C%*\\s%trror:%m,
      \%+C%*[^:]%trror:%m,
      \%C%*\\s%tarning:%m,
      \%C%m
```

Jikes(TM) produces a single-line error message when invoked with the option "+E", and can be matched with the following:

```
:setl efm=%f:%l:%v:%*\\d:%*\\d:%*\\s%m
```

errorformat-javac

This 'errorformat' has been reported to work well for javac, which outputs a line with "^" to indicate the column of the error:

```
:setl efm=%A%f:%l:\ %m,%-Z%p^,%-C%.%#
```

or:

```
:setl efm=%A%f:%l:\ %m,%+Z%p^,%+C%.%#,%-G%.%#
```

Here is an alternative from Michael F. Lamb for Unix that filters the errors first:

```
:setl errorformat=%Z%f:%l:\ %m,%A%p^,%-G%*[^sl]%.%#
:setl makeprg=javac\ %:S\ 2>&1\ \\\|\ vim-javac-filter
```

You need to put the following in "vim-javac-filter" somewhere in your path (e.g., in ~/bin) and make it executable:

```
#!/bin/sed -f
/\^$/s/\t/\ /g;/:[0-9]\+:/{h;d};/^[ \t]*\^/G;
```

In English, that sed script:

- Changes single tabs to single spaces and

- Moves the line with the filename, line number, error message to just after

  the pointer line. That way, the unused error text between doesn't break vim's notion of a "multi-line message" and also doesn't force us to include it as a "continuation of a multi-line message."

errorformat-ant

For ant (https://jakarta.apache.org/) the above errorformat has to be modified
to honour the leading [javac] in front of each javac output line:

```
:set efm=%A\ %#[javac]\ %f:%l:\ %m,%-Z\ %#[javac]\ %p^,%-C%.%#
```

The 'errorformat' can also be configured to handle ant together with either
javac or jikes. If you're using jikes, you should tell ant to use jikes' +E
command line switch which forces jikes to generate one-line error messages.
This is what the second line (of a build.xml file) below does:

```
<property name = "build.compiler"       value = "jikes"/>
<property name = "build.compiler.emacs" value = "true"/>
```

The 'errorformat' which handles ant with both javac and jikes is:

```
:set efm=\ %#[javac]\ %#%f:%l:%c:%*\\d:%*\\d:\ %t%[%^:]%#:%m,
         \%A\ %#[javac]\ %f:%l:\ %m,%-Z\ %#[javac]\ %p^,%-C%.%#
```

errorformat-jade

parsing jade (see http://www.jclark.com/) errors is simple:

```
:set efm=jade:%f:%l:%c:%t:%m
```

errorformat-LaTeX

The following is an example how an 'errorformat' string can be specified
for the (La)TeX typesetting system which displays error messages over
multiple lines. The output of ":clist" and ":cc" etc. commands displays
multi-lines in a single line, leading white space is removed.
It should be easy to adopt the above LaTeX errorformat to any compiler output
consisting of multi-line errors.

The commands can be placed in a vimrc file or some other Vim script file,
e.g. a script containing LaTeX related stuff which is loaded only when editing
LaTeX sources.
Make sure to copy all lines of the example (in the given order), afterwards
remove the comment lines. For the '\' notation at the start of some lines see
line-continuation.

First prepare 'makeprg' such that LaTeX will report multiple
errors; do not stop when the first error has occurred:

```
:set makeprg=latex\ \\\\nonstopmode\ \\\\input\\{$*}
```

Start of multi-line error messages:

```
:set efm=%E!\ LaTeX\ %trror:\ %m,
       \%E!\ %m,
```

Start of multi-line warning messages; the first two also
include the line number. Meaning of some regular expressions:

- "%.%#" (.*)   matches a (possibly empty) string

- "%*\\d" ("\d\+") matches a number

```
\%+WLaTeX\ %.%#Warning:\ %.%#line\ %l%.%#,
\%+W%.%#\ at\ lines\ %l--%*\\d,
\%WLaTeX\ %.%#Warning:\ %m,
```

Possible continuations of error/warning messages; the first
one also includes the line number:

```
\%Cl.%l\ %m,
\%+C\ \ %m.,
\%+C%.%#-%.%#,
\%+C%.%#[]%.%#,
\%+C[]%.%#,
\%+C%.%#%[{}\\]%.%#,
\%+C<%.%#>%.%#,
\%C\ \ %m,
```

Lines that match the following patterns do not contain any
important information; do not include them in messages:

```
\%-GSee\ the\ LaTeX%m,
\%-GType\ \ H\ <return>%m,
\%-G\ ...%.%#,
\%-G%.%#\ (C)\ %.%#,
\%-G(see\ the\ transcript%.%#),
```

Generally exclude any empty or whitespace-only line from
being displayed:

```
\%-G\\s%#,
```

The LaTeX output log does not specify the names of erroneous
source files per line; rather they are given globally,
enclosed in parentheses.
The following patterns try to match these names and store
them in an internal stack. The patterns possibly scan over
the same input line (one after another), the trailing "%r"
conversion indicates the "rest" of the line that will be
parsed in the next go until the end of line is reached.

Overread a file name enclosed in '('...')'; do not push it
on a stack since the file apparently does not contain any
error:

```
\%+O(%f)%r,
```

Push a file name onto the stack. The name is given after '(':

```
\%+P(%f%r,
\%+P\ %\\=(%f%r,
\%+P%*[^()](%f%r,
\%+P[%\\d%[^()]%#(%f%r,
```

Pop the last stored file name when a ')' is scanned:

```
\%+Q)%r,
\%+Q%*[^()])%r,
\%+Q[%\\d%*[^()])%r
```

Note that in some cases file names in the LaTeX output log cannot be parsed
properly. The parser might have been messed up by unbalanced parentheses
then. The above example tries to catch the most relevant cases only.
You can customize the given setting to suit your own purposes, for example,
all the annoying "Overfull ..." warnings could be excluded from being
recognized as an error.
Alternatively to filtering the LaTeX compiler output, it is also possible
to directly read the *.log file that is produced by the [La]TeX compiler.
This contains even more useful information about possible error causes.
However, to properly parse such a complex file, an external filter should
be used. See the description further above how to make such a filter known
by Vim.

10. Customizing the quickfix window quickfix-window-function

The default format for the lines displayed in the quickfix window and location
list window is:

```
<filename>|<lnum> col <col>|<text>
```

The values displayed in each line correspond to the "bufnr", "lnum", "col" and
"text" fields returned by the getqflist() function.

For some quickfix/location lists, the displayed text needs to be customized.
For example, if only the filename is present for a quickfix entry, then the
two "|" field separator characters after the filename are not needed. Another
use case is to customize the path displayed for a filename. By default, the
complete path (which may be too long) is displayed for files which are not
under the current directory tree. The file path may need to be simplified to
a common parent directory.

The displayed text can be customized by setting the 'quickfixtextfunc' option
to a Vim function. This function will be called with a dict argument and
should return a List of strings to be displayed in the quickfix or location
list window. The dict argument will have the following fields:

```
    quickfix    set to 1 when called for a quickfix list and 0 when called for
                a location list.
    winid       for a location list, set to the id of the window with the
                location list.  For a quickfix list, set to 0.  Can be used in
                getloclist() to get the location list entry.
    id          quickfix or location list identifier
    start_idx   index of the first entry for which text should be returned
    end_idx     index of the last entry for which text should be returned
```

The function should return a single line of text to display in the quickfix
window for each entry from start_idx to end_idx. The function can obtain
information about the entries using the getqflist() function and specifying
the quickfix list identifier "id". For a location list, getloclist() function
can be used with the "winid" argument. If an empty list is returned, then the
default format is used to display all the entries. If an item in the returned
list is an empty string, then the default format is used to display the
corresponding entry.

If a quickfix or location list specific customization is needed, then the
'quickfixtextfunc' attribute of the list can be set using the setqflist() or
setloclist() function. This overrides the global 'quickfixtextfunc' option.

The example below displays the list of old files (v:oldfiles) in a quickfix
window. As there is no line, column number and error text information
associated with each entry, the 'quickfixtextfunc' function returns only the
filename.
Example:

```
" create a quickfix list from v:oldfiles
call setqflist([], ' ', {'lines' : v:oldfiles, 'efm' : '%f',
                                    \ 'quickfixtextfunc' : 'QfOldFiles'})
func QfOldFiles(info)
    " get information about a range of quickfix entries
    let items = getqflist({'id' : a:info.id, 'items' : 1}).items
    let l = []
    for idx in range(a:info.start_idx - 1, a:info.end_idx - 1)
        " use the simplified file name
      call add(l, fnamemodify(bufname(items[idx].bufnr), ':p:.'))
    endfor
    return l
endfunc
```

Quick reference guide

Contents

tag	subject	tag	subject
Q_ct	list of help files	Q_re	Repeating commands
Q_lr	motion: Left-right	Q_km	Key mapping
Q_ud	motion: Up-down	Q_ab	Abbreviations
Q_tm	motion: Text object	Q_op	Options
Q_pa	motion: Pattern searches	Q_ur	Undo/Redo commands
Q_ma	motion: Marks	Q_et	External commands
Q_vm	motion: Various	Q_qf	Quickfix commands
Q_ta	motion: Using tags	Q_vc	Various commands
Q_sc	Scrolling	Q_ce	Ex: Command-line editing
Q_in	insert: Inserting text	Q_ra	Ex: Ranges
Q_ai	insert: Keys	Q_ex	Ex: Special characters
Q_ss	insert: Special keys	Q_st	Starting Vim
Q_di	insert: Digraphs	Q_ed	Editing a file
Q_si	insert: Special inserts	Q_fl	Using the argument list
Q_de	change: Deleting text	Q_wq	Writing and quitting
Q_cm	change: Copying and moving	Q_ac	Automatic commands
Q_ch	change: Changing text	Q_wi	Multi-window commands
Q_co	change: Complex	Q_bu	Buffer list commands
Q_vi	Visual mode	Q_sy	Syntax highlighting
Q_to	Text objects	Q_gu	GUI commands
		Q_fo	Folding

Q_lr — Left-right motions

N is used to indicate an optional count that can be given before the command.

h	N	h	left (also: CTRL-H, <BS>, or <Left> key)
l	N	l	right (also: <Space> or <Right> key)
0		0	to first character in the line (also: <Home> key)
^		^	to first non-blank character in the line
$	N	$	to the next EOL (end of line) position (also: <End> key)
g0		g0	to first character in screen line (differs from "0" when lines wrap)
g^		g^	to first non-blank character in screen line (differs from "^" when lines wrap)
g$	N	g$	to last character in screen line (differs from "$" when lines wrap)
gm		gm	to middle of the screen line
gM		gM	to middle of the line
bar	N	|	to column N (default: 1)
f	N	f{char}	to the Nth occurrence of {char} to the right
F	N	F{char}	to the Nth occurrence of {char} to the left
t	N	t{char}	till before the Nth occurrence of {char} to the right
T	N	T{char}	till before the Nth occurrence of {char} to the left
;	N	;	repeat the last "f", "F", "t", or "T" N times
,	N	,	repeat the last "f", "F", "t", or "T" N times in opposite direction

Q_ud — Up-down motions

k	N	k	up N lines (also: CTRL-P and <Up>)
j	N	j	down N lines (also: CTRL-J, CTRL-N, <NL>, and <Down>)
-	N	-	up N lines, on the first non-blank character
+	N	+	down N lines, on the first non-blank character (also: CTRL-M and <CR>)
_	N	_	down N-1 lines, on the first non-blank character
G	N	G	goto line N (default: last line), on the first non-blank character
gg	N	gg	goto line N (default: first line), on the first non-blank character
N%	N	%	goto line N percentage down in the file; N must be given, otherwise it is the % command
gk	N	gk	up N screen lines (differs from "k" when line wraps)
gj	N	gj	down N screen lines (differs from "j" when line wraps)

Q_tm — Text object motions

w	N	w	N words forward
W	N	W	N blank-separated WORDs forward
e	N	e	forward to the end of the Nth word
E	N	E	forward to the end of the Nth blank-separated WORD
b	N	b	N words backward
B	N	B	N blank-separated WORDs backward
ge	N	ge	backward to the end of the Nth word
gE	N	gE	backward to the end of the Nth blank-separated WORD
)	N	)	N sentences forward
(	N	(	N sentences backward
}	N	}	N paragraphs forward
{	N	{	N paragraphs backward
]]	N	]]	N sections forward, at start of section
[[	N	[[	N sections backward, at start of section
][	N	][	N sections forward, at end of section
[]	N	[]	N sections backward, at end of section
[(	N	[(	N times back to unclosed '('

```
[{      N  [{           N times back to unclosed '{'
[m      N  [m           N times back to start of method (for Java)
[M      N  [M           N times back to end of method (for Java)
])      N  ])           N times forward to unclosed ')'
]}      N  ]}           N times forward to unclosed '}'
]m      N  ]m           N times forward to start of method (for Java)
]M      N  ]M           N times forward to end of method (for Java)
[#      N  [#           N times back to unclosed "#if" or "#else"
]#      N  ]#           N times forward to unclosed "#else" or "#endif"
[star   N  [*           N times back to start of comment "/*"
]star   N  ]*           N times forward to end of comment "*/"
```

Q_pa **Pattern searches**

```
/       N  /{pattern}[/[offset]]<CR>
                        search forward for the Nth occurrence of {pattern}
?       N  ?{pattern}[?[offset]]<CR>
                        search backward for the Nth occurrence of {pattern}
/<CR>   N  /<CR>        repeat last search, in the forward direction
?<CR>   N  ?<CR>        repeat last search, in the backward direction
n       N  n            repeat last search
N       N  N            repeat last search, in opposite direction
star    N  *            search forward for the identifier under the cursor
#       N  #            search backward for the identifier under the cursor
gstar   N  g*           like "*", but also find partial matches
g#      N  g#           like "#", but also find partial matches
gd         gd           goto local declaration of identifier under the cursor
gD         gD           goto global declaration of identifier under the cursor
```

pattern Special characters in search patterns

meaning	magic	nomagic
matches any single character	.	\.
matches start of line	^	^
matches <EOL>	$	$
matches start of word	\<	\<
matches end of word	\>	\>
matches a single char from the range	[a-z]	\[a-z]
matches a single char not in the range	[^a-z]	\[^a-z]
matches an identifier char	\i	\i
idem but excluding digits	\I	\I
matches a keyword character	\k	\k
idem but excluding digits	\K	\K
matches a file name character	\f	\f
idem but excluding digits	\F	\F
matches a printable character	\p	\p
idem but excluding digits	\P	\P
matches a white space character	\s	\s
matches a non-white space character	\S	\S

```
                              matches <Esc>    \e      \e
                              matches <Tab>    \t      \t
                               matches <CR>    \r      \r
                               matches <BS>    \b      \b

    matches 0 or more of the preceding atom    *       \*
    matches 1 or more of the preceding atom    \+      \+
       matches 0 or 1 of the preceding atom    \=      \=
       matches 2 to 5 of the preceding atom    \{2,5}  \{2,5}
                 separates two alternatives    \|      \|
              group a pattern into an atom     \(\)    \(\)
```

search-offset Offsets allowed after search command

```
    [num]       [num] lines downwards, in column 1
    +[num]      [num] lines downwards, in column 1
    -[num]      [num] lines upwards, in column 1
    e[+num]     [num] characters to the right of the end of the match
    e[-num]     [num] characters to the left of the end of the match
    s[+num]     [num] characters to the right of the start of the match
    s[-num]     [num] characters to the left of the start of the match
    b[+num]     [num] identical to s[+num] above (mnemonic: begin)
    b[-num]     [num] identical to s[-num] above (mnemonic: begin)
    ;{search-command}   execute {search-command} next
```

Q_ma **Marks and motions**

```
m       m{a-zA-Z}       mark current position with mark {a-zA-Z}
`a      `{a-z} go to mark {a-z} within current file
`A      `{A-Z} go to mark {A-Z} in any file
`0      `{0-9} go to the position where Vim was previously exited
``      ``              go to the position before the last jump
`quote  `"              go to the position when last editing this file
`[      `[              go to the start of the previously operated or put text
`]      `]              go to the end of the previously operated or put text
```

`<	`<		go to the start of the (previous) Visual area
`>	`>		go to the end of the (previous) Visual area
``	``		go to the position of the last change in this file
'	'{a-zA-Z0-9[]'"<>.}
			same as `, but on the first non-blank in the line
:marks	:marks	print the active marks
CTRL-O	N CTRL-O	go to Nth older position in jump list
CTRL-I	N CTRL-I	go to Nth newer position in jump list
:ju	:ju[mps]	print the jump list

Q_vm — Various motions

%	%		find the next brace, bracket, comment, or "#if"/
			"#else"/"#endif" in this line and go to its match
H	N H		go to the Nth line in the window, on the first
			non-blank
M	M		go to the middle line in the window, on the first
			non-blank
L	N L		go to the Nth line from the bottom, on the first
			non-blank

go	N go		go to Nth byte in the buffer
:go	:[range]go[to] [off]	go to [off] byte in the buffer

Q_ta — Using tags

:ta	:ta[g][!] {tag}	jump to tag {tag}
:ta	:[count]ta[g][!]	jump to [count]'th newer tag in tag list
CTRL-]	CTRL-]		jump to the tag under cursor, unless changes
			have been made
:ts	:ts[elect][!] [tag]	list matching tags and select one to jump to
:tjump	:tj[ump][!] [tag]	jump to tag [tag] or select from list when
			there are multiple matches
:ltag	:lt[ag][!] [tag]	jump to tag [tag] and add matching tags to the
			location list

:tags	:tags		print tag list
CTRL-T	N CTRL-T		jump back from Nth older tag in tag list
:po	:[count]po[p][!]	jump back from [count]'th older tag in tag list
:tnext	:[count]tn[ext][!]	jump to [count]'th next matching tag
:tp	:[count]tp[revious][!]	jump to [count]'th previous matching tag
:tr	:[count]tr[ewind][!]	jump to [count]'th matching tag
:tl	:tl[ast][!]		jump to last matching tag

:ptag	:pt[ag] {tag}	open a preview window to show tag {tag}
CTRL-W_}	CTRL-W }		like CTRL-] but show tag in preview window
:pts	:pts[elect]		like ":tselect" but show tag in preview window
:ptjump	:ptj[ump]		like ":tjump" but show tag in preview window
:pclose	:pc[lose]		close tag preview window
CTRL-W_z	CTRL-W z		close tag preview window

Q_sc — Scrolling

CTRL-E		N CTRL-E	window N lines downwards (default: 1)
CTRL-D		N CTRL-D	window N lines Downwards (default: 1/2 window)
CTRL-F		N CTRL-F	window N pages Forwards (downwards)
CTRL-Y		N CTRL-Y	window N lines upwards (default: 1)
CTRL-U		N CTRL-U	window N lines Upwards (default: 1/2 window)
CTRL-B		N CTRL-B	window N pages Backwards (upwards)
z<CR>		z<CR> or zt	redraw, current line at top of window
z.		z. or zz	redraw, current line at center of window
z-		z- or zb	redraw, current line at bottom of window

These only work when 'wrap' is off:
zh		N zh		scroll screen N characters to the right
zl		N zl		scroll screen N characters to the left
zH		N zH		scroll screen half a screenwidth to the right
zL		N zL		scroll screen half a screenwidth to the left

Q_in — Inserting text

a	N a	append text after the cursor (N times)
A	N A	append text at the end of the line (N times)
i	N i	insert text before the cursor (N times) (also: <Insert>)
I	N I	insert text before the first non-blank in the line (N times)
gI	N gI	insert text in column 1 (N times)
o	N o	open a new line below the current line, append text (N times)
O	N O	open a new line above the current line, append text (N times)
:startinsert	:star[tinsert][!]	start Insert mode, append when [!] used
:startreplace	:startr[eplace][!]	start Replace mode, at EOL when [!] used

in Visual block mode:
v_b_I	I	insert the same text in front of all the selected lines
v_b_A	A	append the same text after all the selected lines

Q_ai — Insert mode keys

insert-index	alphabetical index of Insert mode commands

leaving Insert mode:
i_<Esc>		<Esc>		end Insert mode, back to Normal mode
i_CTRL-C		CTRL-C		like <Esc>, but do not use an abbreviation
i_CTRL-O		CTRL-O {command}	execute {command} and return to Insert mode

moving around:

i_<Up>	cursor keys	move cursor left/right/up/down
i_<S-Left>	shift-left/right	one word left/right
i_<S-Up>	shift-up/down	one screenful backward/forward
i_<End>	<End>	cursor after last character in the line
i_<Home>	<Home>	cursor to first character in the line

Q_ss — Special keys in Insert mode

i_CTRL-V	CTRL-V {char}..	insert character literally, or enter decimal byte value
i_<NL>	<NL> or <CR> or CTRL-M or CTRL-J	begin new line
i_CTRL-E	CTRL-E	insert the character from below the cursor
i_CTRL-Y	CTRL-Y	insert the character from above the cursor
i_CTRL-A	CTRL-A	insert previously inserted text
i_CTRL-@	CTRL-@	insert previously inserted text and stop Insert mode
i_CTRL-R	CTRL-R {register}	insert the contents of a register
i_CTRL-N	CTRL-N	insert next match of identifier before the cursor
i_CTRL-P	CTRL-P	insert previous match of identifier before the cursor
i_CTRL-X	CTRL-X ...	complete the word before the cursor in various ways
i_<BS>	<BS> or CTRL-H	delete the character before the cursor
i_<Del>	<Del>	delete the character under the cursor
i_CTRL-W	CTRL-W	delete word before the cursor
i_CTRL-U	CTRL-U	delete all entered characters in the current line
i_CTRL-T	CTRL-T	insert one shiftwidth of indent in front of the current line
i_CTRL-D	CTRL-D	delete one shiftwidth of indent in front of the current line
i_0_CTRL-D	0 CTRL-D	delete all indent in the current line
i_^_CTRL-D	^ CTRL-D	delete all indent in the current line, restore indent in next line

Q_di — Digraphs

:dig	:dig[raphs]	show current list of digraphs
:dig	:dig[raphs] {char1}{char2} {number} ...	add digraph(s) to the list

In Insert or Command-line mode:

i_CTRL-K	CTRL-K {char1} {char2}	enter digraph
i_digraph	{char1} <BS> {char2}	enter digraph if 'digraph' option set

Q_si — Special inserts

:r	:r [file]	insert the contents of [file] below the cursor
:r!	:r! {command}	insert the standard output of {command} below the cursor

Q_de — Deleting text

x	N	x	delete N characters under and after the cursor
<Del>	N	<Del>	delete N characters under and after the cursor
X	N	X	delete N characters before the cursor
d	N	d{motion}	delete the text that is moved over with {motion}
v_d		{visual}d	delete the highlighted text
dd	N	dd	delete N lines
D	N	D	delete to the end of the line (and N-1 more lines)
J	N	J	join N-1 lines (delete <EOL>s)
v_J		{visual}J	join the highlighted lines
gJ	N	gJ	like "J", but without inserting spaces
v_gJ		{visual}gJ	like "{visual}J", but without inserting spaces
:d	:[range]d [x]		delete [range] lines [into register x]

Q_cm — Copying and moving text

quote	"{char}		use register {char} for the next delete, yank, or put
:reg	:reg		show the contents of all registers
:reg	:reg {arg}		show the contents of registers mentioned in {arg}
y	N	y{motion}	yank the text moved over with {motion} into a register
v_y		{visual}y	yank the highlighted text into a register
yy	N	yy	yank N lines into a register
Y	N	Y	yank N lines into a register **Note:** Mapped to "y$" by default. default-mappings
p	N	p	put a register after the cursor position (N times)
P	N	P	put a register before the cursor position (N times)
]p	N	]p	like p, but adjust indent to current line
[p	N	[p	like P, but adjust indent to current line
gp	N	gp	like p, but leave cursor after the new text
gP	N	gP	like P, but leave cursor after the new text

Q_ch — Changing text

```
r          N  r{char}    replace N characters with {char}
gr         N  gr{char}   replace N characters without affecting layout
R          N  R          enter Replace mode (repeat the entered text N times)
gR         N  gR         enter virtual Replace mode: Like Replace mode but
                            without affecting layout
v_b_r         {visual}r{char}
                         in Visual block mode: Replace each char of the
                            selected text with {char}

        (change = delete text and enter Insert mode)
c          N  c{motion}  change the text that is moved over with {motion}
v_c           {visual}c  change the highlighted text
cc         N  cc         change N lines
S          N  S          change N lines
C          N  C          change to the end of the line (and N-1 more lines)
s          N  s          change N characters
v_b_c         {visual}c  in Visual block mode: Change each of the selected
                            lines with the entered text
v_b_C         {visual}C  in Visual block mode: Change each of the selected
                            lines until end-of-line with the entered text

~          N  ~          switch case for N characters and advance cursor
v_~           {visual}~  switch case for highlighted text
v_u           {visual}u  make highlighted text lowercase
v_U           {visual}U  make highlighted text uppercase
g~            g~{motion} switch case for the text that is moved over with
                            {motion}
gu            gu{motion} make the text that is moved over with {motion}
                            lowercase
gU            gU{motion} make the text that is moved over with {motion}
                            uppercase
v_g?          {visual}g? perform rot13 encoding on highlighted text
g?            g?{motion} perform rot13 encoding on the text that is moved over
                            with {motion}

CTRL-A  N  CTRL-A       add N to the number at or after the cursor
CTRL-X  N  CTRL-X       subtract N from the number at or after the cursor

<          N  <{motion}  move the lines that are moved over with {motion} one
                            shiftwidth left
<<         N  <<         move N lines one shiftwidth left
>          N  >{motion}  move the lines that are moved over with {motion} one
                            shiftwidth right
>>         N  >>         move N lines one shiftwidth right
gq         N  gq{motion} format the lines that are moved over with {motion} to
                            'textwidth' length
:ce        :[range]ce[nter] [width]
                         center the lines in [range]
:le        :[range]le[ft] [indent]
                         left-align the lines in [range] (with [indent])
:ri        :[range]ri[ght] [width]
                         right-align the lines in [range]
```

Q_co — Complex changes

```
!          N  !{motion}{command}<CR>
                         filter the lines that are moved over through {command}
!!         N  !!{command}<CR>
                         filter N lines through {command}
v_!           {visual}!{command}<CR>
                         filter the highlighted lines through {command}
:range!   :[range]! {command}<CR>
                         filter [range] lines through {command}
=          N  ={motion}
                         filter the lines that are moved over through 'equalprg'
==         N  ==         filter N lines through 'equalprg'
v_=           {visual}=
                         filter the highlighted lines through 'equalprg'
:s         :[range]s[ubstitute]/{pattern}/{string}/[g][c]
                         substitute {pattern} by {string} in [range] lines;
                            with [g], replace all occurrences of {pattern};
                            with [c], confirm each replacement
:s         :[range]s[ubstitute] [g][c]
                         repeat previous ":s" with new range and options
&             &          Repeat previous ":s" on current line without options
:ret       :[range]ret[ab][!] [tabstop]
                         set 'tabstop' to new value and adjust white space
                            accordingly
```

Q_vi — Visual mode

```
visual-index    list of Visual mode commands.

v          v             start highlighting characters  }  move cursor and use
V          V             start highlighting linewise    }  operator to affect
CTRL-V     CTRL-V start highlighting blockwise   }  highlighted text
v_o        o             exchange cursor position with start of highlighting
gv         gv            start highlighting on previous visual area
v_v        v             highlight characters or stop highlighting
v_V        V             highlight linewise or stop highlighting
v_CTRL-V CTRL-V highlight blockwise or stop highlighting
```

Q_to — Text objects (only in Visual mode or after an operator)

```
v_aw        N  aw        Select "a word"
v_iw        N  iw        Select "inner word"
v_aW        N  aW        Select "a WORD"
v_iW        N  iW        Select "inner WORD"
v_as        N  as        Select "a sentence"
v_is        N  is        Select "inner sentence"
v_ap        N  ap        Select "a paragraph"
v_ip        N  ip        Select "inner paragraph"
v_al           al        Select all lines
v_il           il        Select "inner line"
v_ab        N  ab        Select "a block" (from "[(" to "])")
v_ib        N  ib        Select "inner block" (from "[(" to "])")
v_aB        N  aB        Select "a Block" (from [{ to ]})
v_iB        N  iB        Select "inner Block" (from [{ to ]})
v_a>        N  a>        Select "a <> block"
v_i>        N  i>        Select "inner <> block"
v_at        N  at        Select "a tag block" (from <aaa> to </aaa>)
v_it        N  it        Select "inner tag block" (from <aaa> to </aaa>)
v_a'        N  a'        Select "a single quoted string"
v_i'        N  i'        Select "inner single quoted string"
v_aquote N  a"  Select "a double quoted string"
v_iquote N  i"  Select "inner double quoted string"
v_a`        N  a`        Select "a backward quoted string"
v_i`        N  i`        Select "inner backward quoted string"
```

Q_re — Repeating commands

```
.           N  .         repeat last change (with count replaced with N)
q              q{a-z}    record typed characters into register {a-z}
q              q{A-Z}    record typed characters, appended to register {a-z}
q              q         stop recording
Q              Q         replay last recorded macro
@           N  @{a-z}    execute the contents of register {a-z} (N times)
@@          N  @@           repeat previous @{a-z} (N times)
:@          :@{a-z}      execute the contents of register {a-z} as an Ex
                            command
:@@         :@@          repeat previous :@{a-z}
:g          :[range]g[lobal]/{pattern}/[cmd]
                         execute Ex command [cmd] (default: ":p") on the lines
                            within [range] where {pattern} matches
:g          :[range]g[lobal]!/{pattern}/[cmd]
                         execute Ex command [cmd] (default: ":p") on the lines
                            within [range] where {pattern} does NOT match
:so         :so[urce] {file}
                         read Ex commands from {file}
:so         :so[urce]! {file}
                         read Vim commands from {file}
:sl         :sl[eep] [sec]
                         don't do anything for [sec] seconds
gs          N  gs        goto Sleep for N seconds
```

Q_km — Key mapping

```
:map        :ma[p] {lhs} {rhs}     map {lhs} to {rhs} in Normal and Visual mode
:map!       :ma[p]! {lhs} {rhs}  map {lhs} to {rhs} in Insert and Command-line
                                    mode
:noremap    :no[remap][!] {lhs} {rhs}
                                 same as ":map", no remapping for this {rhs}
:unmap      :unm[ap] {lhs}         remove the mapping of {lhs} for Normal and
                                    Visual mode
:unmap!     :unm[ap]! {lhs}        remove the mapping of {lhs} for Insert and
                                    Command-line mode
:map_l      :ma[p] [lhs]   list mappings (starting with [lhs]) for
                                    Normal and Visual mode
:map_l!     :ma[p]! [lhs]          list mappings (starting with [lhs]) for
                                    Insert and Command-line mode
:cmap       :cmap/:cunmap/:cnoremap
                                 like ":map!"/":unmap!"/":noremap!" but for
                                    Command-line mode only
:imap       :imap/:iunmap/:inoremap
                                 like ":map!"/":unmap!"/":noremap!" but for
                                    Insert mode only
:nmap       :nmap/:nunmap/:nnoremap
                                 like ":map"/":unmap"/":noremap" but for
                                    Normal mode only
:vmap       :vmap/:vunmap/:vnoremap
                                 like ":map"/":unmap"/":noremap" but for
                                    Visual mode only
:omap       :omap/:ounmap/:onoremap
                                 like ":map"/":unmap"/":noremap" but only for
                                    when an operator is pending
:mapc       :mapc[lear]    remove mappings for Normal and Visual mode
:mapc       :mapc[lear]!   remove mappings for Insert and Cmdline mode
:imapc      :imapc[lear]   remove mappings for Insert mode
:vmapc      :vmapc[lear]   remove mappings for Visual mode
:omapc      :omapc[lear]   remove mappings for Operator-pending mode
:nmapc      :nmapc[lear]   remove mappings for Normal mode
:cmapc      :cmapc[lear]   remove mappings for Cmdline mode
```

:mkexrc	:mk[exrc][!] [file]	write current mappings, abbreviations, and settings to [file] (default: ".exrc"; use ! to overwrite)
:mkvimrc	:mkv[imrc][!] [file]	same as :mkexrc, but with default ".nvimrc"
:mksession	:mks[ession][!] [file]	like ":mkvimrc", but store current files, windows, etc. too, to be able to continue this session later

Q_ab — Abbreviations

:abbreviate	:ab[breviate] {lhs} {rhs}	add abbreviation for {lhs} to {rhs}
:abbreviate	:ab[breviate] {lhs}	show abbr's that start with {lhs}
:abbreviate	:ab[breviate]	show all abbreviations
:unabbreviate	:una[bbreviate] {lhs}	remove abbreviation for {lhs}
:noreabbrev	:norea[bbrev] [lhs] [rhs]	like ":ab", but don't remap [rhs]
:iabbrev	:iab/:iunab/:inoreab	like ":ab", but only for Insert mode
:cabbrev	:cab/:cunab/:cnoreab	like ":ab", but only for Command-line mode
:abclear	:abc[lear]	remove all abbreviations
:cabclear	:cabc[lear]	remove all abbr's for Cmdline mode
:iabclear	:iabc[lear]	remove all abbr's for Insert mode

Q_op — Options

:set	:se[t]	show all modified options
:set	:se[t] all	show all options
:set	:se[t] {option}	set boolean option (switch it on), show string or number option
:set	:se[t] no{option}	reset boolean option (switch it off)
:set	:se[t] inv{option}	invert boolean option
:set	:se[t] {option}={value}	set string/number option to {value}
:set	:se[t] {option}+={value}	append {value} to string option, add {value} to number option
:set	:se[t] {option}-={value}	remove {value} to string option, subtract {value} from number option
:set	:se[t] {option}?	show value of {option}
:set	:se[t] {option}&	reset {option} to its default value
:setlocal	:setl[ocal]	like ":set" but set the local value for options that have one
:setglobal	:setg[lobal]	like ":set" but set the global value of a local option
:options	:opt[ions]	open a new window to view and set options, grouped by functionality, a one line explanation and links to the help

Short explanation of each option: option-list

'allowrevins'	'ari'	allow CTRL-_ in Insert mode
'ambiwidth'	'ambw'	what to do with Unicode chars of ambiguous width
'arabic'	'arab'	for Arabic as a default second language
'arabicshape'	'arshape'	do shaping for Arabic characters
'autochdir'	'acd'	change directory to the file in the current window
'autocomplete'	'ac'	enable automatic completion in insert mode
'autocompletedelay'	'acl'	delay in msec before menu appears after typing
'autocompletetimeout'	'act'	initial decay timeout for autocompletion algorithm
'autoindent'	'ai'	take indent for new line from previous line
'autoread'	'ar'	autom. read file when changed outside of Vim
'autowrite'	'aw'	automatically write file if changed
'autowriteall'	'awa'	as 'autowrite', but works with more commands
'background'	'bg'	"dark" or "light", used for highlight colors
'backspace'	'bs'	how backspace works at start of line
'backup'	'bk'	keep backup file after overwriting a file
'backupcopy'	'bkc'	make backup as a copy, don't rename the file
'backupdir'	'bdir'	list of directories for the backup file
'backupext'	'bex'	extension used for the backup file
'backupskip'	'bsk'	no backup for files that match these patterns
'belloff'	'bo'	do not ring the bell for these reasons
'binary'	'bin'	read/write/edit file in binary mode
'bomb'		prepend a Byte Order Mark to the file
'breakat'	'brk'	characters that may cause a line break
'breakindent'	'bri'	wrapped line repeats indent
'breakindentopt'	'briopt'	settings for 'breakindent'
'browsedir'	'bsdir'	which directory to start browsing in
'bufhidden'	'bh'	what to do when buffer is no longer in window
'buflisted'	'bl'	whether the buffer shows up in the buffer list
'buftype'	'bt'	special type of buffer
'casemap'	'cmp'	specifies how case of letters is changed
'cdhome'	'cdh'	change directory to the home directory by ":cd"
'cdpath'	'cd'	list of directories searched with ":cd"
'cedit'		key used to open the command-line window
'charconvert'	'ccv'	expression for character encoding conversion
'chistory'	'chi'	maximum number of quickfix lists in history
'cindent'	'cin'	do C program indenting
'cinkeys'	'cink'	keys that trigger indent when 'cindent' is set
'cinoptions'	'cino'	how to do indenting when 'cindent' is set
'cinscopedecls'	'cinsd'	words that are recognized by 'cino-g'
'cinwords'	'cinw'	words where 'si' and 'cin' add an indent

'clipboard'	'cb'	use the clipboard as the unnamed register
'cmdheight'	'ch'	number of lines to use for the command-line
'cmdwinheight'	'cwh'	height of the command-line window
'colorcolumn'	'cc'	columns to highlight
'columns'	'co'	number of columns in the display
'comments'	'com'	patterns that can start a comment line
'commentstring'	'cms'	template for comments; used for fold marker
'complete'	'cpt'	specify how Insert mode completion works
'completefunc'	'cfu'	function to be used for Insert mode completion
'completeopt'	'cot'	options for Insert mode completion
'completeslash'	'csl'	like 'shellslash' for completion
'completetimeout'	'cto'	initial decay timeout for CTRL-N and CTRL-P
'concealcursor'	'cocu'	whether concealable text is hidden in cursor line
'conceallevel'	'cole'	whether concealable text is shown or hidden
'confirm'	'cf'	ask what to do about unsaved/read-only files
'copyindent'	'ci'	make 'autoindent' use existing indent structure
'cpoptions'	'cpo'	flags for Vi-compatible behavior
'cursorbind'	'crb'	move cursor in window as it moves in other windows
'cursorcolumn'	'cuc'	highlight the screen column of the cursor
'cursorline'	'cul'	highlight the screen line of the cursor
'cursorlineopt'	'culopt'	settings for 'cursorline'
'debug'		set to "msg" to see all error messages
'define'	'def'	pattern to be used to find a macro definition
'delcombine'	'deco'	delete combining characters on their own
'dictionary'	'dict'	list of file names used for keyword completion
'diff'		use diff mode for the current window
'diffanchors'	'dia'	list of {address} to force anchoring of a diff
'diffexpr'	'dex'	expression used to obtain a diff file
'diffopt'	'dip'	options for using diff mode
'digraph'	'dg'	enable the entering of digraphs in Insert mode
'directory'	'dir'	list of directory names for the swap file
'display'	'dy'	list of flags for how to display text
'eadirection'	'ead'	in which direction 'equalalways' works
'encoding'	'enc'	encoding used internally
'endoffile'	'eof'	write CTRL-Z at end of the file
'endofline'	'eol'	write <EOL> for last line in file
'equalalways'	'ea'	windows are automatically made the same size
'equalprg'	'ep'	external program to use for "=" command
'errorbells'	'eb'	ring the bell for error messages
'errorfile'	'ef'	name of the errorfile for the QuickFix mode
'errorformat'	'efm'	description of the lines in the error file
'eventignore'	'ei'	autocommand events that are ignored
'eventignorewin'	'eiw'	autocommand events that are ignored in a window
'expandtab'	'et'	use spaces when <Tab> is inserted
'exrc'	'ex'	read init files in the current directory
'fileencoding'	'fenc'	file encoding for multibyte text
'fileencodings'	'fencs'	automatically detected character encodings
'fileformat'	'ff'	file format used for file I/O
'fileformats'	'ffs'	automatically detected values for 'fileformat'
'fileignorecase'	'fic'	ignore case when using file names
'filetype'	'ft'	type of file, used for autocommands
'fillchars'	'fcs'	characters to use for displaying special items
'findfunc'	'ffu'	function to be called for the :find command
'fixendofline'	'fixeol'	make sure last line in file has <EOL>
'foldclose'	'fcl'	close a fold when the cursor leaves it
'foldcolumn'	'fdc'	width of the column used to indicate folds
'foldenable'	'fen'	set to display all folds open
'foldexpr'	'fde'	expression used when 'foldmethod' is "expr"
'foldignore'	'fdi'	ignore lines when 'foldmethod' is "indent"
'foldlevel'	'fdl'	close folds with a level higher than this
'foldlevelstart'	'fdls'	'foldlevel' when starting to edit a file
'foldmarker'	'fmr'	markers used when 'foldmethod' is "marker"
'foldmethod'	'fdm'	folding type
'foldminlines'	'fml'	minimum number of lines for a fold to be closed
'foldnestmax'	'fdn'	maximum fold depth
'foldopen'	'fdo'	for which commands a fold will be opened
'foldtext'	'fdt'	expression used to display for a closed fold
'follow'		multicursor: motions cascade to all cursors
'formatexpr'	'fex'	expression used with "gq" command
'formatlistpat'	'flp'	pattern used to recognize a list header
'formatoptions'	'fo'	how automatic formatting is to be done
'formatprg'	'fp'	name of external program used with "gq" command
'fsync'	'fs'	whether to invoke fsync() after file write
'grepformat'	'gfm'	format of 'grepprg' output
'grepprg'	'gp'	program to use for ":grep"
'guicursor'	'gcr'	GUI: settings for cursor shape and blinking
'guifont'	'gfn'	GUI: Name(s) of font(s) to be used
'guifontwide'	'gfw'	list of font names for double-wide characters
'guioptions'	'go'	GUI: Which components and options are used
'guitablabel'	'gtl'	GUI: custom label for a tabpage
'guitabtooltip'	'gtt'	GUI: custom tooltip for a tabpage
'helpfile'	'hf'	full path name of the main help file
'helpheight'	'hh'	minimum height of a new help window
'helplang'	'hlg'	preferred help languages
'hidden'	'hid'	don't unload buffer when it is abandoned
'history'	'hi'	number of command-lines that are remembered
'hlsearch'	'hls'	highlight matches with last search pattern
'icon'		let Vim set the text of the window icon
'iconstring'		string to use for the Vim icon text
'ignorecase'	'ic'	ignore case in search patterns

'imcmdline'	'imc'	use IM when starting to edit a command line
'imdisable'	'imd'	do not use the IM in any mode
'iminsert'	'imi'	use :lmap or IM in Insert mode
'imsearch'	'ims'	use :lmap or IM when typing a search pattern
'include'	'inc'	pattern to be used to find an include file
'includeexpr'	'inex'	expression used to process an include line
'incsearch'	'is'	highlight match while typing search pattern
'indentexpr'	'inde'	expression used to obtain the indent of a line
'indentkeys'	'indk'	keys that trigger indenting with 'indentexpr'
'infercase'	'inf'	adjust case of match for keyword completion
'isfname'	'isf'	characters included in file names and pathnames
'isident'	'isi'	characters included in identifiers
'iskeyword'	'isk'	characters included in keywords
'joinspaces'	'js'	two spaces after a period with a join command
'jumpoptions'	'jop'	specifies how jumping is done
'keymap'	'kmp'	name of a keyboard mapping
'keymodel'	'km'	enable starting/stopping selection with keys
'keywordprg'	'kp'	program to use for the "K" command
'langmap'	'lmap'	alphabetic characters for other language mode
'langmenu'	'lm'	language to be used for the menus
'langremap'	'lrm'	do apply 'langmap' to mapped characters
'laststatus'	'ls'	tells when last window has status lines
'lazyredraw'	'lz'	don't redraw while executing macros
'lhistory'	'lhi'	maximum number of location lists in history
'linebreak'	'lbr'	wrap long lines at a blank
'lines'		number of lines in the display
'linespace'	'lsp'	number of pixel lines to use between characters
'lisp'		automatic indenting for Lisp
'lispoptions'	'lop'	changes how Lisp indenting is done
'lispwords'	'lw'	words that change how lisp indenting works
'list'		show <Tab> and <EOL>
'listchars'	'lcs'	characters for displaying in list mode
'loadplugins'	'lpl'	load plugin scripts when starting up
'makeef'	'mef'	name of the errorfile for ":make"
'makeencoding'	'menc'	encoding of external make/grep commands
'makeprg'	'mp'	program to use for the ":make" command
'matchpairs'	'mps'	pairs of characters that "%" can match
'matchtime'	'mat'	tenths of a second to show matching paren
'maxcombine'	'mco'	maximum nr of combining characters displayed
'maxfuncdepth'	'mfd'	maximum recursive depth for user functions
'maxmapdepth'	'mmd'	maximum recursive depth for mapping
'maxmempattern'	'mmp'	maximum memory (in Kbyte) used for pattern search
'menuitems'	'mis'	maximum number of items in a menu
'mkspellmem'	'msm'	memory used before :mkspell compresses the tree
'modeline'	'ml'	recognize modelines at start or end of file
'modelineexpr'	'mle'	allow setting expression options from a modeline
'modelines'	'mls'	number of lines checked for modelines
'modifiable'	'ma'	changes to the text are not possible
'modified'	'mod'	buffer has been modified
'more'		pause listings when the whole screen is filled
'mouse'		enable the use of mouse clicks
'mousefocus'	'mousef'	keyboard focus follows the mouse
'mousehide'	'mh'	hide mouse pointer while typing
'mousemodel'	'mousem'	changes meaning of mouse buttons
'mousemoveevent'	'mousemev'	report mouse moves with <MouseMove>
'mousescroll'		amount to scroll by when scrolling with a mouse
'mouseshape'	'mouses'	shape of the mouse pointer in different modes
'mousetime'	'mouset'	max time between mouse double-click
'nrformats'	'nf'	number formats recognized for CTRL-A command
'number'	'nu'	print the line number in front of each line
'numberwidth'	'nuw'	number of columns used for the line number
'omnifunc'	'ofu'	function for filetype-specific completion
'opendevice'	'odev'	allow reading/writing devices on MS-Windows
'operatorfunc'	'opfunc'	function to be called for g@ operator
'packpath'	'pp'	list of directories used for packages
'paragraphs'	'para'	nroff macros that separate paragraphs
'patchexpr'	'pex'	expression used to patch a file
'patchmode'	'pm'	keep the oldest version of a file
'path'	'pa'	list of directories searched with "gf" et.al.
'preserveindent'	'pi'	preserve the indent structure when reindenting
'previewheight'	'pvh'	height of the preview window
'previewwindow'	'pvw'	identifies the preview window
'pumheight'	'ph'	maximum number of items to show in the popup menu
'pumwidth'	'pw'	minimum width of the popup menu
'pyxversion'	'pyx'	Python version used for pyx* commands
'quoteescape'	'qe'	escape characters used in a string
'readonly'	'ro'	disallow writing the buffer
'redrawtime'	'rdt'	timeout for 'hlsearch' and :match highlighting
'regexpengine'	're'	default regexp engine to use
'relativenumber'	'rnu'	show relative line number in front of each line
'report'		threshold for reporting nr. of lines changed
'revins'	'ri'	inserting characters will work backwards
'rightleft'	'rl'	window is right-to-left oriented
'rightleftcmd'	'rlc'	commands for which editing works right-to-left
'ruler'	'ru'	show cursor line and column in the status line
'rulerformat'	'ruf'	custom format for the ruler
'runtimepath'	'rtp'	list of directories used for runtime files
'scroll'	'scr'	lines to scroll with CTRL-U and CTRL-D
'scrollbind'	'scb'	scroll in window as other windows scroll
'scrolljump'	'sj'	minimum number of lines to scroll

'scrolloff'	'so'	minimum nr. of lines above and below cursor
'scrolloffpad'	'sop'	vertically center cursor at end of file
'scrollopt'	'sbo'	how 'scrollbind' should behave
'sections'	'sect'	nroff macros that separate sections
'secure'		secure mode for reading .vimrc in current dir
'selection'	'sel'	what type of selection to use
'selectmode'	'slm'	when to use Select mode instead of Visual mode
'sessionoptions'	'ssop'	options for :mksession
'shada'	'sd'	use shada file upon startup and exiting
'shell'	'sh'	name of shell to use for external commands
'shellcmdflag'	'shcf'	flag to shell to execute one command
'shellpipe'	'sp'	string to put output of ":make" in error file
'shellquote'	'shq'	quote character(s) for around shell command
'shellredir'	'srr'	string to put output of filter in a temp file
'shellslash'	'ssl'	use forward slash for shell file names
'shelltemp'	'stmp'	whether to use a temp file for shell commands
'shellxescape'	'sxe'	characters to escape when 'shellxquote' is (
'shellxquote'	'sxq'	like 'shellquote', but include redirection
'shiftround'	'sr'	round indent to multiple of shiftwidth
'shiftwidth'	'sw'	number of spaces to use for (auto)indent step
'shortmess'	'shm'	list of flags, reduce length of messages
'showbreak'	'sbr'	string to use at the start of wrapped lines
'showcmd'	'sc'	show (partial) command somewhere
'showcmdloc'	'sloc'	where to show (partial) command
'showfulltag'	'sft'	show full tag pattern when completing tag
'showmatch'	'sm'	briefly jump to matching bracket if insert one
'showmode'	'smd'	message on status line to show current mode
'showtabline'	'stal'	tells when the tabpages line is displayed
'sidescroll'	'ss'	minimum number of columns to scroll horizontal
'sidescrolloff'	'siso'	min. nr. of columns to left and right of cursor
'signcolumn'	'scl'	when and how to display the sign column
'smartcase'	'scs'	no ignore case when pattern has uppercase
'smartindent'	'si'	smart autoindenting for C programs
'smarttab'	'sta'	<Tab> in leading whitespace indents by 'shiftwidth'
'smoothscroll'	'sms'	scroll by screen lines when 'wrap' is set
'softtabstop'	'sts'	number of columns between two soft tab stops
'spell'		enable spell checking
'spellcapcheck'	'spc'	pattern to locate end of a sentence
'spellfile'	'spf'	files where zg and zw store words
'spelllang'	'spl'	language(s) to do spell checking for
'spelloptions'	'spo'	options for spell checking
'spellsuggest'	'sps'	method(s) used to suggest spelling corrections
'splitbelow'	'sb'	new window from split is below the current one
'splitkeep'	'spk'	determines scroll behavior for split windows
'splitright'	'spr'	new window is put right of the current one
'startofline'	'sol'	commands move cursor to first non-blank in line
'statuscolumn'	'stc'	custom format for the status column
'statusline'	'stl'	custom format for the status line
'suffixes'	'su'	suffixes that are ignored with multiple match
'suffixesadd'	'sua'	suffixes added when searching for a file
'swapfile'	'swf'	whether to use a swapfile for a buffer
'switchbuf'	'swb'	sets behavior when switching to another buffer
'synmaxcol'	'smc'	maximum column to find syntax items
'syntax'	'syn'	syntax to be loaded for current buffer
'tabclose'	'tcl'	which tabpage to focus when closing a tab
'tabline'	'tal'	custom format for the console tabpages line
'tabpagemax'	'tpm'	maximum number of tabpages for -p and "tab all"
'tabstop'	'ts'	number of columns between two tab stops
'tagbsearch'	'tbs'	use binary searching in tags files
'tagcase'	'tc'	how to handle case when searching in tags files
'tagfunc'	'tfu'	function to get list of tag matches
'taglength'	'tl'	number of significant characters for a tag
'tagrelative'	'tr'	file names in tag file are relative
'tags'	'tag'	list of file names used by the tag command
'tagstack'	'tgst'	push tags onto the tag stack
'term'		name of the terminal
'termbidi'	'tbidi'	terminal takes care of bi-directionality
'termguicolors'	'tgc'	enable 24-bit RGB color in the TUI
'textwidth'	'tw'	maximum width of text that is being inserted
'thesaurus'	'tsr'	list of thesaurus files for keyword completion
'thesaurusfunc'	'tsrfu'	function to be used for thesaurus completion
'tildeop'	'top'	tilde command "~" behaves like an operator
'timeout'	'to'	time out on mappings and key codes
'timeoutlen'	'tm'	time out time in milliseconds
'title'		let Vim set the title of the window
'titlelen'		percentage of 'columns' used for window title
'titleold'		old title, restored when exiting
'titlestring'		string to use for the Vim window title
'ttimeout'		time out on mappings
'ttimeoutlen'	'ttm'	time out time for key codes in milliseconds
'ttytype'	'tty'	alias for 'term'
'undodir'	'udir'	where to store undo files
'undofile'	'udf'	save undo information in a file
'undolevels'	'ul'	maximum number of changes that can be undone
'undoreload'	'ur'	max nr of lines to save for undo on a buffer reload
'updatecount'	'uc'	after this many characters flush swap file
'updatetime'	'ut'	after this many milliseconds flush swap file
'varsofttabstop'	'vsts'	a list of number of columns between soft tab stops
'vartabstop'	'vts'	a list of number of columns between tab stops
'verbose'	'vbs'	give informative messages

'verbosefile'	'vfile'	file to write messages in
'viewdir'	'vdir'	directory where to store files with :mkview
'viewoptions'	'vop'	specifies what to save for :mkview
'virtualedit'	've'	when to use virtual editing
'visualbell'	'vb'	use visual bell instead of beeping
'warn'		warn for shell command when buffer was changed
'whichwrap'	'ww'	allow specified keys to cross line boundaries
'wildchar'	'wc'	command-line character for wildcard expansion
'wildcharm'	'wcm'	like 'wildchar' but also works when mapped
'wildignore'	'wig'	files matching these patterns are not completed
'wildignorecase'	'wic'	ignore case when completing file names
'wildmenu'	'wmnu'	use menu for command line completion
'wildmode'	'wim'	mode for 'wildchar' command-line expansion
'wildoptions'	'wop'	specifies how command line completion is done
'winaltkeys'	'wak'	when the windows system handles ALT keys
'window'	'wi'	nr of lines to scroll for CTRL-F and CTRL-B
'winfixbuf'	'wfb'	keep window focused on a single buffer
'winfixheight'	'wfh'	keep window height when opening/closing windows
'winfixwidth'	'wfw'	keep window width when opening/closing windows
'winheight'	'wh'	minimum number of lines for the current window
'winhighlight'	'winhl'	window-local highlighting
'winminheight'	'wmh'	minimum number of lines for any window
'winminwidth'	'wmw'	minimal number of columns for any window
'winwidth'	'wiw'	minimal number of columns for current window
'wrap'		long lines wrap and continue on the next line
'wrapmargin'	'wm'	chars from the right where wrapping starts
'wrapscan'	'ws'	searches wrap around the end of the file
'write'		writing to a file is allowed
'writeany'	'wa'	write to file with no need for "!" override
'writebackup'	'wb'	make a backup before overwriting a file
'writedelay'	'wd'	delay this many msec for each char (for debug)

Q_ur — Undo/Redo commands

u	N	u	undo last N changes
CTRL-R	N	CTRL-R	redo last N undone changes
U		U	restore last changed line

Q_et — External commands

:!	:!{command}	execute {command} with a shell
K	K	lookup keyword under the cursor with 'keywordprg' program (default: "man")

Q_qf — Quickfix commands

:cc	:cc [nr]	display error [nr] (default is the same again)
:cnext	:cn	display the next error
:cprevious	:cp	display the previous error
:clist	:cl	list all errors
:cfile	:cf	read errors from the file 'errorfile'
:cgetbuffer	:cgetb	like :cbuffer but don't jump to the first error
:cgetfile	:cg	like :cfile but don't jump to the first error
:cgetexpr	:cgete	like :cexpr but don't jump to the first error
:caddfile	:caddf	add errors from the error file to the current quickfix list
:caddexpr	:cad	add errors from an expression to the current quickfix list
:cbuffer	:cb	read errors from text in a buffer
:cexpr	:cex	read errors from an expression
:cquit	:cq	quit without writing and return error code (to the compiler)
:make	:make [args]	start make, read errors, and jump to first error
:grep	:gr[ep] [args]	execute 'grepprg' to find matches and jump to the first one

Q_vc — Various commands

CTRL-L	CTRL-L	clear and redraw the screen
CTRL-G	CTRL-G	show current file name (with path) and cursor position
ga	ga	show ascii value of character under cursor in decimal, hex, and octal
g8	g8	for utf-8 encoding: show byte sequence for character under cursor in hex
g_CTRL-G	g CTRL-G	show cursor column, line, and character position
CTRL-C	CTRL-C	during searches: Interrupt the search
<Del>	<Del>	while entering a count: delete last character
:version	:ve[rsion]	show version information
:normal	:norm[al][!] {commands}	execute Normal mode commands
:exmode	1q:	switch to Ex-mode

:redir	:redir >{file}	redirect messages to {file}
:silent	:silent[!] {command}	execute {command} silently
:confirm	:confirm {command}	quit, write, etc., asking about unsaved changes or read-only files
:browse	:browse {command}	open/read/write file, using a file selection dialog

Q_ce — Command-line editing

c_<Esc>	<Esc>	abandon command-line (if 'wildchar' is <Esc>, type it twice)
c_CTRL-V	CTRL-V {char}	insert {char} literally
c_CTRL-V	CTRL-V {number}	enter decimal value of character (up to three digits)
c_CTRL-K	CTRL-K {char1} {char2}	enter digraph (See Q_di)
c_CTRL-R	CTRL-R {register}	insert the contents of a register
c_<Left>	<Left>/<Right>	cursor left/right
c_<S-Left>	<S-Left>/<S-Right>	cursor one word left/right
c_CTRL-B	CTRL-B/CTRL-E	cursor to beginning/end of command-line
c_<BS>	<BS>	delete the character in front of the cursor
c_<Del>	<Del>	delete the character under the cursor
c_CTRL-W	CTRL-W	delete the word in front of the cursor
c_CTRL-U	CTRL-U	remove all characters between the cursor and beginning of command-line
c_<Up>	<Up>/<Down>	recall older/newer command-line that starts with current command
c_<S-Up>	<S-Up>/<S-Down>	recall older/newer command-line from history
c_CTRL-G	CTRL-G	next match when 'incsearch' is active
c_CTRL-T	CTRL-T	previous match when 'incsearch' is active
:history	:his[tory]	show older command-lines

Context-sensitive completion on the command-line:

c_wildchar	'wildchar' (default: <Tab>)	do completion on the pattern in front of the cursor; if there are multiple matches, beep and show the first one; further 'wildchar' will show the next ones
c_CTRL-D	CTRL-D	list all names that match the pattern in front of the cursor
c_CTRL-A	CTRL-A	insert all names that match pattern in front of cursor
c_CTRL-L	CTRL-L	insert longest common part of names that match pattern
c_CTRL-N	CTRL-N	after 'wildchar' with multiple matches: go to next match
c_CTRL-P	CTRL-P	after 'wildchar' with multiple matches: go to previous match

Q_ra — Ex ranges

:range	,	separates two line numbers
:range	;	idem, set cursor to the first line number before interpreting the second one
:range	{number}	an absolute line number
:range	.	the current line
:range	$	the last line in the file
:range	%	equal to 1,$ (the entire file)
:range	*	equal to '<,'> (visual area)
:range	't	position of mark t
:range	/{pattern}[/]	the next line where {pattern} matches
:range	?{pattern}[?]	the previous line where {pattern} matches
:range	+[num]	add [num] to the preceding line number (default: 1)
:range	-[num]	subtract [num] from the preceding line number (default: 1)

Q_ex — Special Ex characters

:bar	|	separates two commands (not for ":global" and ":!")
:quote	"	begins comment
:_%	%	current file name (only where a file name is expected)
:_#	#[num]	alternate file name [num] (only where a file name is expected)

Note: The next seven are typed literally; these are not special keys!

:<abuf>	<abuf>	buffer number, for use in an autocommand (only where a file name is expected)
:<afile>	<afile>	file name, for use in an autocommand (only where a file name is expected)
:<amatch>	<amatch>	what matched with the pattern, for use in an autocommand (only where a file name is expected)
:<cword>	<cword>	word under the cursor (only where a file name is expected)
:<cWORD>	<cWORD>	WORD under the cursor (only where a file name is expected) (see WORD)
:<cfile>	<cfile>	file name under the cursor (only where a file name is expected)
:<script>	<script>	file name of a ":source"d file, within that file (only where a file name is expected)

After "%", "#", "<cfile>", "<script>" or "<afile>"

::p	:p	full path
::h	:h	head (file name removed)
::t	:t	tail (file name only)
::r	:r	root (extension removed)

```
               ::e         :e          extension
               ::s         :s/{pat}/{repl}/    substitute {pat} with {repl}
```

Q_st — Starting Vim

```
-file      vim [options] {file} ..      start editing one or more files
--         vim [options] -              read file from stdin
-tag       vim [options] -t {tag}       edit the file associated with {tag}
-qf        vim [options] -q [fname]     start editing in QuickFix mode,
                                           display the first error

        Most useful Vim arguments (for full list see startup-options)

-+      +[num]              put the cursor at line [num] (default: last line)
-+c     +{command}          execute {command} after loading the file
-+/     +/{pat} {file} ..   put the cursor at the first occurrence of {pat}
-e      -e                  Ex-mode
-R      -R                  Read-only mode, implies -n
-m      -m                  modifications not allowed (resets 'write' option)
-d      -d                  diff-mode
-b      -b                  binary mode
-l      -l                  lisp mode
-A      -A                  Arabic mode ('arabic' is set)
-H      -H                  Hebrew mode (Hebrew keymap & 'rightleft' are set)
-V      -V                  Verbose, give informative messages
-r      -r                  give list of swap files
-r      -r {file} ..        recover aborted edit session
-n      -n                  do not create a swap file
-o      -o [num]            open [num] windows (default: one for each file)
-s      -s {scriptin}       first read commands from the file {scriptin}
-w      -w {scriptout}      write typed chars to file {scriptout} (append)
-W      -W {scriptout}      write typed chars to file {scriptout} (overwrite)
-u      -u {vimrc}          read inits from {vimrc} instead of other inits
-i      -i {shada}          read info from {shada} instead of other files
---     --                  end of options, other arguments are file names
--help    --help            show list of arguments and exit
--version --version         show version info and exit
--       -                  read file from stdin
```

Q_ed — Editing a file

```
           Without !: Fail if changes have been made to the current buffer.
              With !: Discard any changes to the current buffer.
:edit_f  :e[dit][!] {file}      edit {file}
:edit    :e[dit][!]             reload the current file
:enew    :ene[w][!]             edit a new, unnamed buffer
:find    :fin[d][!] {file}      find {file} in 'path' and edit it

CTRL-^   N  CTRL-^              edit alternate file N (equivalent to ":e #N")
gf          gf  or ]f           edit the file whose name is under the cursor
:pwd     :pwd                   print the current directory name
:cd      :cd [path]             change the current directory to [path]
:cd-     :cd -          back to previous current directory
:file    :f[ile]                print the current file name and the cursor
                                   position
:file    :f[ile] {name} set the current file name to {name}
:files   :files         show alternate file names
```

Q_fl — Using the argument list — argument-list

```
:args      :ar[gs]              print the argument list, with the current file
                                   in "[]"
:all       :all  or :sall       open a window for every file in the arg list
:wn        :wn[ext][!]          write file and edit next file
:wn        :wn[ext][!] {file}   write to {file} and edit next file, unless
                                   {file} exists; With !, overwrite existing
                                   file
:wN        :wN[ext][!] [file]   write file and edit previous file
```

in current window	in new window	action
:argument :argu[ment] N	:sar[gument] N	edit file N
:next :n[ext]	:sn[ext]	edit next file
:next_f :n[ext] {arglist}	:sn[ext] {arglist}	define new arg list and edit first file
:Next :N[ext]	:sN[ext]	edit previous file
:first :fir[st]	:sfir[st]	edit first file
:last :la[st]	:sla[st]	edit last file

Q_wq — Writing and quitting

```
:w         :[range]w[rite][!]            write to the current file
:w_f       :[range]w[rite] {file}        write to {file}, unless it already
                                           exists
:w_f       :[range]w[rite]! {file}       write to {file}.  Overwrite an existing
                                           file
:w_a       :[range]w[rite][!] >>         append to the current file
:w_a       :[range]w[rite][!] >> {file}  append to {file}
:w_c       :[range]w[rite] !{cmd}        execute {cmd} with [range] lines as
                                           standard input
```

```
:up       :[range]up[date][!]          write to current file if modified
:wall     :wa[ll][!]                   write all changed buffers

:q        :q[uit]              quit current buffer, unless changes have been
                                  made; Exit Vim when there are no other
                                  non-help buffers
:q        :q[uit]!             quit current buffer always, discard any
                                  changes.  Exit Vim when there are no other
                                  non-help buffers
:qa       :qa[ll]              exit Vim, unless changes have been made
:qa       :qa[ll]!             exit Vim always, discard any changes
:cq       :cq                  quit without writing and return error code

:wq       :wq[!]               write the current file and exit
:wq       :wq[!] {file}        write to {file} and exit
:xit      :x[it][!] [file]     like ":wq" but write only when changes have
                                  been made
ZZ           ZZ                same as ":x"
ZQ           ZQ                same as ":q!"
:xall     :xa[ll][!]  or :wqall[!]
                               write all changed buffers and exit

:stop     :st[op][!]           suspend Vim or start new shell; if 'aw' option
                                  is set and [!] not given write the buffer
CTRL-Z     CTRL-Z              same as ":stop"
```

Q_ac — Automatic Commands

```
shada-file      read registers, marks, history at startup, save when exiting.

:rshada         :rsh[ada] [file]        read info from ShaDa file [file]
:rshada         :rsh[ada]! [file]       idem, overwrite existing info
:wshada         :wsh[ada] [file]        add info to ShaDa file [file]
:wshada         :wsh[ada]! [file]       write info to ShaDa file [file]

modeline        Automatic option setting when editing a file

modeline        vim:{set-arg}: ..       In the first and last lines of the
                                        file (see 'ml' option), {set-arg} is
                                        given as an argument to ":set"

autocommand     Automatic execution of commands on certain events.

:autocmd        :au                       list all autocommands
:autocmd        :au {event}               list all autocommands for {event}
:autocmd        :au {event} {pat}         list all autocommands for {event}
                                          with {pat}
:autocmd        :au {event} {pat} {cmd}   enter new autocommands for {event}
                                          with {pat}
:autocmd        :au!                      remove all autocommands
:autocmd        :au! {event}              remove all autocommands for {event}
:autocmd        :au! * {pat}              remove all autocommands for {pat}
:autocmd        :au! {event} {pat}        remove all autocommands for {event}
                                          with {pat}
:autocmd        :au! {event} {pat} {cmd}  remove all autocommands for {event}
                                          with {pat} and enter new one
```

Q_wi — Multi-window commands

```
CTRL-W_s        CTRL-W s  or  :split    split window into two parts
:split_f        :split {file}           split window and edit {file} in one of
                                           them
:vsplit         :vsplit {file}          same, but split vertically
:vertical       :vertical {cmd}         make {cmd} split vertically

:sfind          :sf[ind] {file}         split window, find {file} in 'path'
                                           and edit it
:terminal       :terminal {cmd}         open a terminal window
CTRL-W_l        CTRL-W ]                split window and jump to tag under
                                           cursor
CTRL-W_f        CTRL-W f                split window and edit file name under
                                           the cursor
CTRL-W_^        CTRL-W ^                split window and edit alternate file
CTRL-W_n        CTRL-W n  or  :new      create new empty window
CTRL-W_q        CTRL-W q  or  :q[uit]   quit editing and close window
CTRL-W_c        CTRL-W c  or  :clo[se]  make buffer hidden and close window
CTRL-W_o        CTRL-W o  or  :on[ly]   make current window only one on the
                                           screen

CTRL-W_j        CTRL-W j                move cursor to window below
CTRL-W_k        CTRL-W k                move cursor to window above
CTRL-W_CTRL-W   CTRL-W CTRL-W           move cursor to window below (wrap)
CTRL-W_W        CTRL-W W                move cursor to window above (wrap)
CTRL-W_t        CTRL-W t                move cursor to top window
CTRL-W_b        CTRL-W b                move cursor to bottom window
CTRL-W_p        CTRL-W p                move cursor to previous active window

CTRL-W_r        CTRL-W r                rotate windows downwards
CTRL-W_R        CTRL-W R                rotate windows upwards
CTRL-W_x        CTRL-W x                exchange current window with next one

CTRL-W_=        CTRL-W =                make all windows equal height & width
CTRL-W_-        CTRL-W -                decrease current window height
CTRL-W_+        CTRL-W +                increase current window height
CTRL-W__        CTRL-W _                set current window height (default:
                                           very high)
```

CTRL-W_<	CTRL-W <	decrease current window width
CTRL-W_>	CTRL-W >	increase current window width
CTRL-W_bar	CTRL-W |	set current window width (default:
		widest possible)

Q_bu Buffer list commands

:buffers	:buffers or :files	list all known buffer and file names

:ball	:ball or :sball	edit all args/buffers
:unhide	:unhide or :sunhide	edit all loaded buffers

:badd	:badd {fname}	add file name {fname} to the list
:bunload	:bunload[!] [N]	unload buffer [N] from memory
:bdelete	:bdelete[!] [N]	unload buffer [N] and delete it from
		the buffer list

	in current window	**in new window**	
:buffer	:[N]buffer [N]	:[N]sbuffer [N]	to arg/buf N
:bnext	:[N]bnext [N]	:[N]sbnext [N]	to Nth next arg/buf
:bNext	:[N]bNext [N]	:[N]sbNext [N]	to Nth previous arg/buf
:bprevious	:[N]bprevious [N]	:[N]sbprevious [N]	to Nth previous arg/buf
:bfirst	:bfirst	:sbfirst	to first arg/buf
:blast	:blast	:sblast	to last arg/buf
:bmodified	:[N]bmod [N]	:[N]sbmod [N]	to Nth modified buf

Q_sy Syntax Highlighting

:syn-on	:syntax on	start using syntax highlighting
:syn-off	:syntax off	stop using syntax highlighting

:syn-keyword	:syntax keyword {group-name} {keyword} ..
		add a syntax keyword item
:syn-match	:syntax match {group-name} {pattern} ...
		add syntax match item
:syn-region	:syntax region {group-name} {pattern} ...
		add syntax region item
:syn-sync	:syntax sync [ccomment | lines {N} | ...]
		tell syntax how to sync
:syntax	:syntax [list]	list current syntax items
:syn-clear	:syntax clear	clear all syntax info

:highlight	:highlight clear	clear all highlight info
:highlight	:highlight {group-name} {key}={arg} ..
		set highlighting for {group-name}

:filetype	:filetype on	switch on file type detection, without
		syntax highlighting
:filetype	:filetype plugin indent on
		switch on file type detection, with
		automatic indenting and settings

Q_gu GUI commands

:menu	:menu	list all menus
:menu	:menu {mpath}	list menus starting with {mpath}
:menu	:menu {mpath} {rhs}	add menu {mpath}, giving {rhs}
:menu	:menu {pri} {mpath} {rhs}
		idem, with priorities {pri}
:menu	:menu ToolBar.{name} {rhs}
		add toolbar item, giving {rhs}
:tmenu	:tmenu {mpath} {text}	add tooltip to menu {mpath}
:unmenu	:unmenu {mpath}	remove menu {mpath}

Q_fo Folding

'foldmethod'	set foldmethod=manual	manual folding
	set foldmethod=indent	folding by indent
	set foldmethod=expr	folding by 'foldexpr'
	set foldmethod=syntax	folding by syntax regions
	set foldmethod=marker	folding by 'foldmarker'

zf	zf{motion}	operator: Define a fold manually
:fold	:{range}fold	define a fold for {range} lines
zd	zd	delete one fold under the cursor
zD	zD	delete all folds under the cursor

zo	zo	open one fold under the cursor
zO	zO	open all folds under the cursor
zc	zc	close one fold under the cursor
zC	zC	close all folds under the cursor

zm	zm	fold more: decrease 'foldlevel'
zM	zM	close all folds: make 'foldlevel' zero
zr	zr	reduce folding: increase 'foldlevel'
zR	zR	open all folds: make 'foldlevel' max.

zn	zn	fold none: reset 'foldenable'
zN	zN	fold normal set 'foldenable'
zi	zi	invert 'foldenable'

Recovery after a crash

You have spent several hours typing in that text that has to be finished
next morning, and then disaster strikes: Your computer crashes.

DON'T PANIC!

You can recover most of your changes from the files that Vim uses to store
the contents of the file. Mostly you can recover your work with one command:
vim -r filename

1. The swap file swap-file

Vim stores the things you changed in a swap file. Using the original file
you started from plus the swap file you can mostly recover your work.

You can see the name of the current swap file being used with the command:

:sw[apname] :sw :swapname

Or you can use the swapname() function, which also allows for seeing the
swap file name of other buffers.

The name of the swap file is normally the same as the file you are editing,
with the extension ".swp".

- On Unix, a '.' is prepended to swap file names in the same directory as the edited file. This avoids that the swap file shows up in a directory listing.
- If this file already exists (e.g., when you are recovering from a crash) a warning is given and another extension is used, ".swo", ".swn", etc.
- An existing file will never be overwritten.
- The swap file is deleted as soon as Vim stops editing the file.

E326

Technical: If the ".swp" file name already exists, the last character is
decremented until there is no file with that name or ".saa" is
reached. In the last case, no swap file is created.

By setting the 'directory' option you can place the swap file in another place
than where the edited file is.
Advantages:

- You will not pollute the directories with ".swp" files.
- When the 'directory' is on another partition, reduce the risk of damaging the file system where the file is (in a crash).

Disadvantages:

- You can get name collisions from files with the same name but in different directories (although Vim tries to avoid that by comparing the path name). This will result in bogus ATTENTION warning messages.
- When you use your home directory, and somebody else tries to edit the same file, that user will not see your swap file and will not get the ATTENTION warning message.

If you want to put swap files in a fixed place, put a command resembling the
following ones in your vimrc:

```
:set dir=~/tmp          (for Unix)
:set dir=c:\\tmp        (for Win32)
```

This is also very handy when editing files on floppy. Of course you will have
to create that "tmp" directory for this to work!

For read-only files, a swap file is not used right away. The swap file is
created only when making changes.

The 'swapfile' option can be reset to avoid creating a swapfile. And the
:noswapfile modifier can be used to not create a swapfile for a new buffer.

:nos[wapfile] {command} :nos :noswapfile
Execute {command}. If it contains a command that loads a new
buffer, it will be loaded without creating a swapfile and the
'swapfile' option will be reset. If a buffer already had a
swapfile it is not removed and 'swapfile' is not reset.

Detecting an existing swap file

You can find this in the user manual, section 11.3.

W325

The default SwapExists handler (default-autocmds) skips the E325 prompt
(and automatically chooses "(E)dit") if the swapfile owner process is still
running and owned by the current user. This presumes that you normally don't
want to be bothered with the ATTENTION message just because you happen to

edit the same file from multiple Nvim instances. In the worst case (a system crash) there will be more than one swapfile for the file; use :recover to inspect all of its swapfiles.

Updating the swapfile

The swap file is updated after typing 200 characters or when you have not typed anything for four seconds. This only happens if the buffer was changed, not when you only moved around. The reason why it is not kept up to date all the time is that this would slow down normal work too much. You can change the 200 character count with the 'updatecount' option. You can set the time with the 'updatetime' option. The time is given in milliseconds. After writing to the swap file Vim syncs the file to disk.

If the writing to the swap file is not wanted, it can be switched off by setting the 'updatecount' option to 0. The same is done when starting Vim with the "-n" option. Writing can be switched back on by setting the 'updatecount' option to non-zero. Swap files will be created for all buffers when doing this. But when setting 'updatecount' to zero, the existing swap files will not be removed, it will only affect files that will be opened after this.

If you want to make sure that your changes are in the swap file use this command:

:pre :preserve E313 E314

:pre[serve] Write all text for the current buffer into its swap file. The original file is no longer needed for recovery.

A Vim swap file can be recognized by the first six characters: "b0VIM ". After that comes the version number, e.g., "3.0".

Links and symbolic links

On Unix it is possible to have two names for the same file. This can be done with hard links and with symbolic links (symlinks).

For hard links Vim does not know the other name of the file. Therefore, the name of the swapfile will be based on the name you used to edit the file. There is no check for editing the same file by the other name too, because Vim cannot find the other swapfile (except for searching all of your harddisk, which would be very slow).

For symbolic links Vim resolves the links to find the name of the actual file. The swap file name is based on that name. Thus it doesn't matter by what name you edit the file, the swap file name will normally be the same. However, there are exceptions:

- When the directory of the actual file is not writable the swapfile is put elsewhere.
- When the symbolic links somehow create a loop you get an E773 error message and the unmodified file name will be used. You won't be able to save your file normally.

2. Recovery recovery E308 E311

Basic file recovery is explained in the user manual: usr_11.txt.

Another way to do recovery is to start Vim and use the ":recover" command. This is easy when you start Vim to edit a file and you get the "ATTENTION: Found a swap file ..." message. In this case the single command ":recover" will do the work. You can also give the name of the file or the swap file to the recover command:

:rec :recover E305 E306 E307

:rec[over] [file] Try to recover [file] from the swap file. If [file] is not given use the file name for the current buffer. The current contents of the buffer are lost. This command fails if the buffer was modified.

:rec[over]! [file] Like ":recover", but any changes in the current buffer are lost.

E312 E309 E310 E1364

Vim has some intelligence about what to do if the swap file is corrupt in some way. If Vim has doubt about what it found, it will give an error message and insert lines with "???" in the text. If you see an error message while recovering, search in the file for "???" to see what is wrong. You may want to cut and paste to get the text you need.

The most common remark is "???LINES MISSING". This means that Vim cannot read the text from the original file. This can happen if the system crashed and parts of the original file were not written to disk.

Be sure that the recovery was successful before overwriting the original file or deleting the swap file. It is good practice to write the recovered file elsewhere and run 'diff' to find out if the changes you want are in the recovered file. Or use :DiffOrig.

Once you are sure the recovery is ok delete the swap file. Otherwise, you will continue to get warning messages that the ".swp" file already exists.

Note: Recovering swap files is best-effort. Vim attempts to validate fields and skip corrupted sections, but the swap file format is intended for files you trust. A crafted swap file may trigger parser bugs; such reports are treated as robustness issues rather than security vulnerabilities.

Vim client-server communication

1. Common functionality clientserver

Nvim's RPC functionality allows clients to programmatically control Nvim. Nvim itself takes command-line arguments that cause it to become a client to another Nvim running as a server. These arguments match those provided by Vim's clientserver option.

The following command line arguments are available:

--remote [+{cmd}] {file} ... --remote

Open the file list in a remote Vim. When there is no Vim server, execute locally. Vim allows one init command: +{cmd}. This must be an Ex command that can be followed by "|". It's not yet supported by Nvim. The rest of the command line is taken as the file list. Thus any non-file arguments must come before this. You cannot edit stdin this way --. The remote Vim is raised. If you don't want this use

```
nvim --remote-send "<C-\><C-N>:n filename<CR>"
```

--remote-silent [+{cmd}] {file} ... --remote-silent

As above, but don't complain if there is no server and the file is edited locally.

--remote-tab --remote-tab

Like --remote but open each file in a new tabpage.

--remote-tab-silent --remote-tab-silent

Like --remote-silent but open each file in a new tabpage.

--remote-send {keys} --remote-send

Send {keys} to server and exit. The {keys} are not mapped. Special key names are recognized, e.g., "<CR>" results in a CR character.

--remote-expr {expr} --remote-expr

Evaluate {expr} in server and print the result on stdout.

--remote-ui --remote-ui

Display the UI of the server in the terminal. Fully interactive: keyboard and mouse input are forwarded to the server.

--server {addr} --server

Connect to the named pipe or socket at the given address for executing remote commands. See --listen for specifying an address when starting a server.

Examples

Start an Nvim server listening on a named pipe at '~/.cache/nvim/server.pipe':

```
nvim --listen ~/.cache/nvim/server.pipe
```

Edit "file.txt" in an Nvim server listening at '~/.cache/nvim/server.pipe':

```
nvim --server ~/.cache/nvim/server.pipe --remote file.txt
```

This doesn't work, all arguments after --remote will be used as file names:

```
nvim --remote --server ~/.cache/nvim/server.pipe file.txt
```

Tell the remote server to write all files and exit:

```
nvim --server ~/.cache/nvim/server.pipe --remote-send '<C-\><C-N>:wqa<CR>'
```

REMOTE EDITING

The --remote argument will cause a :drop command to be constructed from the rest of the command line and sent as described above.
Note that the --remote and --remote-wait arguments will consume the rest of

the command line. I.e. all remaining arguments will be regarded as filenames.
You can not put options there!

2. Missing functionality

E5600 clientserver-missing

Vim supports additional functionality in clientserver that's not yet
implemented in Nvim. In particular, none of the "wait" variants are supported
yet. The following command line arguments are not yet available:

--remote-wait [+{cmd}] {file} ... --remote-wait

Not yet supported by Nvim.
As --remote, but wait for files to complete
(unload) in remote Vim.

--remote-wait-silent [+{cmd}] {file} ... --remote-wait-silent

Not yet supported by Nvim.
As --remote-wait, but don't complain if there
is no server.

--remote-tab-wait --remote-tab-wait

Not yet supported by Nvim.
Like --remote-wait but open each file in a new
tabpage.

--remote-tab-wait-silent --remote-tab-wait-silent

Not yet supported by Nvim.
Like --remote-wait-silent but open each file
in a new tabpage.

Nvim support for remote plugins

1. Introduction

remote-plugin-intro

Extensibility is a primary goal of Nvim. Any programming language may be used to extend Nvim without changes to Nvim itself. This is achieved with remote plugins, coprocesses that have a direct communication channel (via RPC) with the Nvim process.

Even though these plugins run in separate processes they can call, be called, and receive events just as if the plugin's code were executed in the main process.

2. Plugin hosts

remote-plugin-hosts

While plugins can be implemented as arbitrary programs that communicate directly with the high-level Nvim API and are called via rpcrequest() and rpcnotify(), that is not the best approach. Instead, developers should first check whether a plugin host is available for their chosen programming language.

Plugin hosts are programs that provide a high-level environment for plugins, taking care of most boilerplate involved in defining commands, autocmds, and functions implemented over RPC connections. Hosts are loaded only when one of their registered plugins require it, keeping Nvim's startup as fast as possible, even if many plugins/hosts are installed.

3. Example

remote-plugin-example

The best way to learn about remote plugins is with an example, so let's see what a Python plugin looks like. This plugin exports a command, a function, and an autocmd. The plugin is called 'Limit', and all it does is limit the number of requests made to it. Here's the plugin source code:

```
import pynvim

@pynvim.plugin
class Limit(object):
    def __init__(self, vim):
        self.vim = vim
        self.calls = 0

    @pynvim.command('Cmd', range='', nargs='*', sync=True)
    def command_handler(self, args, range):
        self._increment_calls()
        self.vim.current.line = (
            'Command: Called %d times, args: %s, range: %s' % (self.calls,
                                                               args,
                                                               range))

    @pynvim.autocmd('BufEnter', pattern='*.py', eval='expand("<afile>")',
                    sync=True)
    def autocmd_handler(self, filename):
        self._increment_calls()
        self.vim.current.line = (
            'Autocmd: Called %s times, file: %s' % (self.calls, filename))

    @pynvim.function('Func')
    def function_handler(self, args):
        self._increment_calls()
        self.vim.current.line = (
            'Function: Called %d times, args: %s' % (self.calls, args))

    def _increment_calls(self):
        if self.calls == 5:
            raise Exception('Too many calls!')
        self.calls += 1
```

As can be seen, the plugin is implemented using idiomatic Python (classes, methods, and decorators). The translation between these language-specific idioms to Vimscript occurs while the plugin manifest is being generated (see the next section).

Notice that the exported command and autocmd are defined with the "sync" flag, which affects how Nvim calls the plugin: with "sync" the rpcrequest() function is used, which will block Nvim until the handler function returns a value. Without the "sync" flag, the call is made using a fire and forget approach with rpcnotify(), meaning return values or exceptions raised in the handler function are ignored.

To test the above plugin, it must be saved in "rplugin/python" in a 'runtimepath' directory (~/.config/nvim/rplugin/python/limit.py for example). Then, the remote plugin manifest must be generated with :UpdateRemotePlugins.

4. Remote plugin manifest

remote-plugin-manifest

:UpdateRemotePlugins

Just installing remote plugins to "rplugin/{host}" isn't enough for them to be automatically loaded when required. You must execute :UpdateRemotePlugins

every time a remote plugin is installed, updated, or deleted.

:UpdateRemotePlugins generates the remote plugin manifest, a special Vimscript file containing declarations for all Vimscript entities (commands/autocommands/functions) defined by all remote plugins, with each entity associated with the host and plugin path.

Manifest declarations are just calls to the remote#host#RegisterPlugin function, which takes care of bootstrapping the host as soon as the declared command, autocommand, or function is used for the first time.

The manifest generation step is necessary to keep Nvim's startup fast in situations where a user has remote plugins with different hosts. For example, say a user has three plugins, for Python, Java and .NET hosts respectively. If we were to load all three plugins at startup, then three language runtimes would also be spawned, which could take seconds!

With the manifest, each host will only be loaded when required. Continuing with the example, say the Java plugin is a semantic completion engine for Java code. If it defines the autocommand BufEnter *.java, then the Java host is spawned only when Nvim loads a buffer matching "*.java".

If the explicit call to :UpdateRemotePlugins seems inconvenient, try to see it like this: It's a way to provide IDE capabilities in Nvim while still keeping it fast and lightweight for general use. It's also analogous to the :helptags command.

$NVIM_RPLUGIN_MANIFEST

Unless $NVIM_RPLUGIN_MANIFEST is set the manifest will be written to a file named rplugin.vim at:

Unix
: $XDG_DATA_HOME/nvim/ or ~/.local/share/nvim/

Windows
: $LOCALAPPDATA/nvim/ or ~/AppData/Local/nvim/

Scrolling

These commands move the contents of the window. If the cursor position is moved off of the window, the cursor is moved onto the window (with 'scrolloff' screen lines around it). A page is the number of lines in the window minus two. The mnemonics for these commands may be a bit confusing. Remember that the commands refer to moving the window (the part of the buffer that you see) upwards or downwards in the buffer. When the window moves upwards in the buffer, the text in the window moves downwards on your screen.

See section 03.7 of the user manual for an introduction.

1. Scrolling downwards

scroll-down

The following commands move the edit window (the part of the buffer that you see) downwards (this means that more lines downwards in the text buffer can be seen):

CTRL-E

CTRL-E Scroll window [count] lines downwards in the buffer. The text moves upwards on the screen. Mnemonic: Extra lines.

CTRL-D

CTRL-D Scroll window Downwards in the buffer. The number of lines comes from the 'scroll' option (default: half a screen). If [count] given, first set 'scroll' option to [count]. The cursor is moved the same number of lines down in the file (if possible; when lines wrap and when hitting the end of the file there may be a difference). When the cursor is on the last line of the buffer nothing happens and a beep is produced. See also 'startofline' option.

<S-Down> or <S-Down> <kPageDown>
<PageDown> or <PageDown> CTRL-F
<S-CR> or <S-CR> <S-NL>
<S-+> or SHIFT-+ <S-Plus>

CTRL-F Scroll window [count] pages Forwards (downwards) in the buffer. See also 'startofline' option. When there is only one window the 'window' option might be used.

z+

z+ Without [count]: Redraw with the line just below the window at the top of the window. Put the cursor in that line, at the first non-blank in the line. With [count]: just like "z<CR>".

2. Scrolling upwards

scroll-up

The following commands move the edit window (the part of the buffer that you see) upwards (this means that more lines upwards in the text buffer can be seen):

CTRL-Y

CTRL-Y Scroll window [count] lines upwards in the buffer. The text moves downwards on the screen. **Note:** When using the MS-Windows key bindings CTRL-Y is remapped to redo.

CTRL-U

CTRL-U Scroll window Upwards in the buffer. The number of lines comes from the 'scroll' option (default: half a screen). If [count] given, first set the 'scroll' option to [count]. The cursor is moved the same number of lines up in the file (if possible; when lines wrap and when hitting the end of the file there may be a difference). When the cursor is on the first line of the buffer nothing happens and a beep is produced. See also 'startofline' option.

<S-Up> or <S-Up> <kPageUp>
<PageUp> or <PageUp> CTRL-B
<S--> or <S-Minus> SHIFT-MINUS

CTRL-B Scroll window [count] pages Backwards (upwards) in the buffer. See also 'startofline' option. When there is only one window the 'window' option might be used.

z^

z^ Without [count]: Redraw with the line just above the window at the bottom of the window. Put the cursor in that line, at the first non-blank in the line. With [count]: First scroll the text to put the [count] line at the bottom of the window, then redraw with the line which is now at the top of the window at the

bottom of the window. Put the cursor in that line, at the first non-blank in the line.

3. Scrolling relative to cursor *scroll-cursor*

The following commands reposition the edit window (the part of the buffer that you see) while keeping the cursor on the same line. Note that the 'scrolloff' option may cause context lines to show above and below the cursor.

z<CR>

z<CR> Redraw, line [count] at top of window (default cursor line). Put cursor at first non-blank in the line.

zt

zt Like "z<CR>", but leave the cursor in the same column.

zN<CR>

z{height}<CR> Redraw, make window {height} lines tall. This is useful to make the number of lines small when screen updating is very slow. Cannot make the height more than the physical screen height.

z.

z. Redraw, line [count] at center of window (default cursor line). Put cursor at first non-blank in the line.

zz

zz Like "z.", but leave the cursor in the same column. Careful: If caps-lock is on, this command becomes "ZZ": write buffer and exit!

z-

z- Redraw, line [count] at bottom of window (default cursor line). Put cursor at first non-blank in the line.

zb

zb Like "z-", but leave the cursor in the same column.

4. Scrolling horizontally *scroll-horizontal*

For the following four commands the cursor follows the screen. If the character that the cursor is on is moved off the screen, the cursor is moved to the closest character that is on the screen. The value of 'sidescroll' is not used.

z<Right> or *zl* *z<Right>*

zl Move the view on the text [count] characters to the right, thus scroll the text [count] characters to the left. This only works when 'wrap' is off.

z<Left> or *zh* *z<Left>*

zh Move the view on the text [count] characters to the left, thus scroll the text [count] characters to the right. This only works when 'wrap' is off.

zL

zL Move the view on the text half a screenwidth to the right, thus scroll the text half a screenwidth to the left. This only works when 'wrap' is off.

zH

zH Move the view on the text half a screenwidth to the left, thus scroll the text half a screenwidth to the right. This only works when 'wrap' is off.

For the following two commands the cursor is not moved in the text, only the text scrolls on the screen.

zs

zs Scroll the text horizontally to position the cursor at the start (left side) of the screen. This only works when 'wrap' is off.

ze

ze Scroll the text horizontally to position the cursor at the end (right side) of the screen. This only works when 'wrap' is off.

5. Scrolling synchronously *scroll-binding*

Occasionally, it is desirable to bind two or more windows together such that when one window is scrolled, the other windows are also scrolled. In Vim,

windows can be given this behavior by setting the (window-specific) 'scrollbind' option. When a window that has 'scrollbind' set is scrolled, all other 'scrollbind' windows are scrolled the same amount, if possible. The behavior of 'scrollbind' can be modified by the 'scrollopt' option.

When using the scrollbars or the mouse wheel, the binding only happens when scrolling the window with focus (where the cursor is). You can use this to avoid scroll-binding for a moment without resetting options.

When a window also has the 'diff' option set, the scroll-binding uses the differences between the two buffers to synchronize the position precisely. Otherwise the following method is used.

scrollbind-relative

Each 'scrollbind' window keeps track of its "relative offset", which can be thought of as the difference between the current window's vertical scroll position and the other window's vertical scroll position. When one of the 'scrollbind' windows is asked to vertically scroll past the beginning or end limit of its text, the window no longer scrolls, but remembers how far past the limit it wishes to be. The window keeps this information so that it can maintain the same relative offset, regardless of its being asked to scroll past its buffer's limits.

However, if a 'scrollbind' window that has a relative offset that is past its buffer's limits is given the cursor focus, the other 'scrollbind' windows must jump to a location where the current window's relative offset is valid. This behavior can be changed by clearing the "jump" flag from the 'scrollopt' option.

syncbind :syncbind :sync

:syncbind — Force all 'scrollbind' windows to have the same relative offset. I.e., when any of the 'scrollbind' windows is scrolled to the top of its buffer, all of the 'scrollbind' windows will also be at the top of their buffers.

scrollbind-quickadj

The 'scrollbind' flag is meaningful when using keyboard commands to vertically scroll a window, and is also meaningful when using the vertical scrollbar or the mouse wheel in the window which has the cursor focus. However, when using the vertical scrollbar or the mouse wheel in a window which doesn't have the cursor focus, 'scrollbind' is ignored.
This allows quick adjustment of the relative offset of 'scrollbind' windows.

6. Scrolling with a mouse wheel

scroll-mouse-wheel

When your mouse has a scroll wheel, it should work with Nvim in the GUI and any terminal that has mouse support. By default only vertical scroll wheels are supported, but some GUIs also support horizontal scroll wheels.

Note that horizontal scrolling only works if 'nowrap' is set. Also, unless the "h" flag in 'guioptions' is set, the cursor moves to the longest visible line if the cursor line is about to be scrolled off the screen (similarly to how the horizontal scrollbar works).

You can control the number of lines / columns to scroll by using the 'mousescroll' option. You can also modify the default behavior by mapping the keys. For example, to scroll a page at a time in normal mode:

```
:map <ScrollWheelUp> <C-B>
:map <ScrollWheelDown> <C-F>
```

Scroll keys can also be combined with modifiers such as Shift, Ctrl, and Alt.

When scrolling with a mouse, the window currently under the cursor is scrolled. This allows you to scroll inactive windows. Note that when scroll keys are remapped to keyboard keys, the active window is affected regardless of the current cursor position.

1. Introduction sign-intro signs gutter

When a debugger or other IDE tool is driving an editor it needs to be able to give specific highlights which quickly tell the user useful information about the file. One example of this would be a debugger which had an icon in the left-hand column denoting a breakpoint. Another example might be an arrow representing the Program Counter (PC). The sign features allow both placement of a sign, or icon, in the left-hand side of the window and definition of a highlight which will be applied to that line. Displaying the sign as an image is most likely only feasible in gvim (although Sun Microsystem's dtterm does support this it's the only terminal emulator I know of which does). A text sign and the highlight should be feasible in any color terminal emulator.

Signs and highlights are not useful just for debuggers. There are plugins that use signs to mark build errors or display version control status.

There are two steps in using signs:

1. Define the sign. This specifies the image, text and highlighting. For example, you can define a "break" sign with an image of a stop roadsign and text "!!".
2. Place the sign. This specifies the file and line number where the sign is displayed. A defined sign can be placed several times in different lines and files.

sign-column

When signs are defined for a file, Vim will automatically add a column of two characters to display them in. When the last sign is unplaced the column disappears again. This behavior can be changed with the 'signcolumn' option.

The color of the column is set with the SignColumn highlight group hl-SignColumn. Example to set the color:

```
:highlight SignColumn guibg=darkgrey
```

If 'cursorline' is enabled, then the CursorLineSign highlight group is used hl-CursorLineSign.

sign-identifier

Each placed sign is identified by a number called the sign identifier. This identifier is used to jump to the sign or to remove the sign. The identifier is assigned when placing the sign using the :sign-place command or the sign_place() function. Each sign identifier should be a unique number (per buffer). Placing the same identifier twice will move the previously placed sign. The sign_place() function can be called with a zero sign identifier to allocate the next available identifier.

sign-group

Each placed sign can be assigned to either the global group or a named group. When placing a sign, if a group name is not supplied, or an empty string is used, then the sign is placed in the global group. Otherwise the sign is placed in the named group. The sign identifier is unique within a group. The sign group allows Vim plugins to use unique signs without interfering with other plugins using signs.

sign-priority

Each placed sign is assigned a priority value independently of the sign group. The default priority for a sign is 10, this value can be changed for different signs by specifying a different value at definition time. When multiple signs that each have an icon or text are placed on the same line, signs are ordered with decreasing priority from left to right, up until the maximum width set in 'signcolumn'. Low priority signs that do not fit are hidden. Highest priority signs with highlight attributes are always shown.

When the line on which the sign is placed is deleted, the sign is removed along with it.

2. Commands sign-commands :sig :sign

Here is an example that places a sign "piet", displayed with the text ">>", in line 23 of the current file:

```
:sign define piet text=>> texthl=Search
:exe ":sign place 2 line=23 name=piet file=" .. expand("%:p")
```

And here is the command to delete it again:

```
:sign unplace 2
```

Note that the ":sign" command cannot be followed by another command or a

comment. If you do need that, use the :execute command.

DEFINING A SIGN. :sign-define E255 E160 E612

See sign_define() for the equivalent Vim script function.

:sig[n] define {name} {argument}...
Define a new sign or set attributes for an existing sign.
The {name} can either be a number (all digits) or a name
starting with a non-digit. Leading zeros are ignored, thus
"0012", "012" and "12" are considered the same name.
About 120 different signs can be defined.

Accepted arguments:

icon={bitmap}
Define the file name where the bitmap can be found. Should be
a full path. The bitmap should fit in the place of two
characters. This is not checked. If the bitmap is too big it
will cause redraw problems.

toolkit **supports**

Win32	.bmp, .ico, .cur

priority={prio}
Default priority for the sign, see sign-priority.

linehl={group}
Highlighting group used for the whole line the sign is placed
in. Most useful is defining a background color.

numhl={group}
Highlighting group used for the line number on the line where
the sign is placed. Combines with hl-LineNr,
hl-LineNrAbove, hl-LineNrBelow, and hl-CursorLineNr.

text={text} E239

Define the text that is displayed when there is no icon or the
GUI is not being used. Only printable characters are allowed
and they must occupy one or two display cells.

texthl={group}
Highlighting group used for the text item.

culhl={group}
Highlighting group used for the text item when the cursor is
on the same line as the sign and 'cursorline' is enabled.

Example:

```
:sign define MySign text=>> texthl=Search linehl=DiffText
```

DELETING A SIGN :sign-undefine E155

See sign_undefine() for the equivalent Vim script function.

:sig[n] undefine {name}
Deletes a previously defined sign. If signs with this {name}
are still placed this will cause trouble.

Example:

```
:sign undefine MySign
```

LISTING SIGNS :sign-list E156

See sign_getdefined() for the equivalent Vim script function.

:sig[n] list Lists all defined signs and their attributes.

:sig[n] list {name}
Lists one defined sign and its attributes.

PLACING SIGNS :sign-place E158

See sign_place() for the equivalent Vim script function.

:sig[n] place {id} line={lnum} name={name} file={fname}
Place sign defined as {name} at line {lnum} in file {fname}.
:sign-fname

The file {fname} must already be loaded in a buffer. The
exact file name must be used, wildcards, $ENV and ~ are not
expanded, white space must not be escaped. Trailing white
space is ignored.

The sign is remembered under {id}, this can be used for
further manipulation. {id} must be a number. Placing the
same {id} multiple times will move the sign.

The following optional sign attributes can be specified before
"file=":
group={group} Place sign in sign group {group}
priority={prio} Assign priority {prio} to sign

By default, the sign is placed in the global sign group.

By default, the sign is assigned a default priority of 10,
unless specified otherwise by the sign definition. To assign
a different priority value, use "priority={prio}" to specify a

value. The priority is used to determine the sign that is
displayed when multiple signs are placed on the same line.

Examples:

```
:sign place 5 line=3 name=sign1 file=a.py
:sign place 6 group=g2 line=2 name=sign2 file=x.py
:sign place 9 group=g2 priority=50 line=5
                              \ name=sign1 file=a.py
```

:sig[n] place {id} line={lnum} name={name} [buffer={nr}]
Same, but use buffer {nr}. If the buffer argument is not
given, place the sign in the current buffer.

Example:

```
:sign place 10 line=99 name=sign3
:sign place 10 line=99 name=sign3 buffer=3
```

E885

:sig[n] place {id} name={name} file={fname}
Change the placed sign {id} in file {fname} to use the defined
sign {name}. See remark above about {fname} :sign-fname.
This can be used to change the displayed sign without moving
it (e.g., when the debugger has stopped at a breakpoint).

The optional "group={group}" attribute can be used before
"file=" to select a sign in a particular group. The optional
"priority={prio}" attribute can be used to change the priority
of an existing sign.

Example:

```
:sign place 23 name=sign1 file=/path/to/edit.py
```

:sig[n] place {id} name={name} [buffer={nr}]
Same, but use buffer {nr}. If the buffer argument is not
given, use the current buffer.

Example:

```
:sign place 23 name=sign1
:sign place 23 name=sign1 buffer=7
```

REMOVING SIGNS :sign-unplace E159

See sign_unplace() for the equivalent Vim script function.

:sig[n] unplace {id} file={fname}
Remove the previously placed sign {id} from file {fname}.
See remark above about {fname} :sign-fname.

:sig[n] unplace {id} group={group} file={fname}
Same but remove the sign {id} in sign group {group}.

:sig[n] unplace {id} group=* file={fname}
Same but remove the sign {id} from all the sign groups.

:sig[n] unplace * file={fname}
Remove all placed signs in file {fname}.

:sig[n] unplace * group={group} file={fname}
Remove all placed signs in group {group} from file {fname}.

:sig[n] unplace * group=* file={fname}
Remove all placed signs in all the groups from file {fname}.

:sig[n] unplace {id} buffer={nr}
Remove the previously placed sign {id} from buffer {nr}.

:sig[n] unplace {id} group={group} buffer={nr}
Remove the previously placed sign {id} in group {group} from
buffer {nr}.

:sig[n] unplace {id} group=* buffer={nr}
Remove the previously placed sign {id} in all the groups from
buffer {nr}.

:sig[n] unplace * buffer={nr}
Remove all placed signs in buffer {nr}.

:sig[n] unplace * group={group} buffer={nr}
Remove all placed signs in group {group} from buffer {nr}.

:sig[n] unplace * group=* buffer={nr}
Remove all placed signs in all the groups from buffer {nr}.

:sig[n] unplace {id}
Remove the previously placed sign {id} from all files it
appears in.

:sig[n] unplace {id} group={group}
Remove the previously placed sign {id} in group {group} from
all files it appears in.

:sig[n] unplace {id} group=*
Remove the previously placed sign {id} in all the groups from
all the files it appears in.

:sig[n] unplace *
Remove all placed signs in the global group from all the

files.

:sig[n] unplace * group={group}
Remove all placed signs in group {group} from all the files.

:sig[n] unplace * group=*
Remove all placed signs in all the groups from all the files.

:sig[n] unplace
Remove a placed sign at the cursor position. If multiple signs are placed in the line, then only one is removed.

:sig[n] unplace group={group}
Remove a placed sign in group {group} at the cursor position.

:sig[n] unplace group=*
Remove a placed sign in any group at the cursor position.

LISTING PLACED SIGNS **:sign-place-list**

See sign_getplaced() for the equivalent Vim script function.

:sig[n] place file={fname}
List signs placed in file {fname}.
See remark above about {fname} :sign-fname.

:sig[n] place group={group} file={fname}
List signs in group {group} placed in file {fname}.

:sig[n] place group=* file={fname}
List signs in all the groups placed in file {fname}.

:sig[n] place buffer={nr}
List signs placed in buffer {nr}.

:sig[n] place group={group} buffer={nr}
List signs in group {group} placed in buffer {nr}.

:sig[n] place group=* buffer={nr}
List signs in all the groups placed in buffer {nr}.

:sig[n] place List placed signs in the global group in all files.

:sig[n] place group={group}
List placed signs with sign group {group} in all files.

:sig[n] place group=*
List placed signs in all sign groups in all files.

JUMPING TO A SIGN **:sign-jump** **E157**

See sign_jump() for the equivalent Vim script function.

:sig[n] jump {id} file={fname}
Open the file {fname} or jump to the window that contains {fname} and position the cursor at sign {id}.
See remark above about {fname} :sign-fname.
If the file isn't displayed in window and the current file can not be abandoned this fails.

:sig[n] jump {id} group={group} file={fname}
Same but jump to the sign in group {group}

:sig[n] jump {id} [buffer={nr}] E934

Same, but use buffer {nr}. This fails if buffer {nr} does not have a name. If the buffer argument is not given, use the current buffer.

:sig[n] jump {id} group={group} [buffer={nr}]
Same but jump to the sign in group {group}

3. Functions **sign-functions-details**

See:
- sign_define()
- sign_getdefined()
- sign_getplaced()
- sign_jump()
- sign_place()
- sign_placelist()
- sign_undefine()
- sign_unplace()

- sign_unplacelist()

Spell checking

1. Quick start

spell-quickstart E756

This command switches on spell checking:

```
:setlocal spell spelllang=en_us
```

This switches on the 'spell' option and specifies to check for US English.

The words that are not recognized are highlighted with one of these:

SpellBad	word not recognized	hl-SpellBad
SpellCap	word not capitalised	hl-SpellCap
SpellRare	rare word	hl-SpellRare
SpellLocal	wrong spelling for selected region	hl-SpellLocal

Vim only checks words for spelling, there is no grammar check.

If the 'mousemodel' option is set to "popup" and the cursor is on a badly spelled word or it is "popup_setpos" and the mouse pointer is on a badly spelled word, then the popup menu will contain a submenu to replace the bad word. **Note:** this slows down the appearance of the popup menu.

To search for the next misspelled word:

]s

]s Move to next misspelled word after the cursor.
A count before the command can be used to repeat.
'wrapscan' applies.

[s

[s Like "]s" but search backwards, find the misspelled word before the cursor. Doesn't recognize words split over two lines, thus may stop at words that are not highlighted as bad. Does not stop at word with missing capital at the start of a line.

]S

]S Like "]s" but only stop at bad words, not at rare words or words for another region.

[S

[S Like "]S" but search backwards.

]r

]r Move to next "rare" word after the cursor.
A count before the command can be used to repeat.
'wrapscan' applies.

[r

[r Like "]r" but search backwards, find the "rare" word before the cursor. Doesn't recognize words split over two lines, thus may stop at words that are not highlighted as rare.

To add words to your own word list:

zg

zg Add word under the cursor as a good word to the first name in 'spellfile'. A count may precede the command to indicate the entry in 'spellfile' to be used. A count of two uses the second entry.

In Visual mode the selected characters are added as a word (including white space!).
When the cursor is on text that is marked as badly spelled then the marked text is used.
Otherwise the word under the cursor, separated by non-word characters, is used.

If the word is explicitly marked as bad word in another spell file the result is unpredictable.

zG

zG Like "zg" but add the word to the internal word list internal-wordlist.

zw

zw Like "zg" but mark the word as a wrong (bad) word. If the word already appears in 'spellfile' it is turned into a comment line. See spellfile-cleanup for getting rid of those.

zW

zW Like "zw" but add the word to the internal word list internal-wordlist.

zuw zug zuw

zug Undo zw and zg, remove the word from the entry in 'spellfile'. Count used as with zg.

zuW zuG zuW

zuG Undo zW and zG, remove the word from the internal word list. Count used as with zg.

:spe :spellgood E1280

:[count]spe[llgood] {word}
Add {word} as a good word to 'spellfile', like with zg. Without count the first name is used, with a count of two the second entry, etc.

:spe[llgood]! {word} Add {word} as a good word to the internal word list, like with zG.

:spellw :spellwrong

:[count]spellw[rong] {word}
Add {word} as a wrong (bad) word to 'spellfile', as with zw. Without count the first name is used, with a count of two the second entry, etc.

:spellw[rong]! {word} Add {word} as a wrong (bad) word to the internal word list, like with zW.

:spellra :spellrare

:[count]spellra[re] {word}
Add {word} as a rare word to 'spellfile', similar to zw. Without count the first name is used, with a count of two the second entry, etc.

There are no normal mode commands to mark words as rare as this is a fairly uncommon command and all intuitive commands for this are already taken. If you want you can add mappings with e.g.:

```
nnoremap z?  :exe ':spellrare  ' .. expand('<cWORD>')<CR>
nnoremap z/  :exe ':spellrare! ' .. expand('<cWORD>')<CR>
```

:spellundo, zuw, or zuW can be used to undo this.

:spellra[re]! {word} Add {word} as a rare word to the internal word list, similar to zW.

:[count]spellu[ndo] {word} :spellu :spellundo
Like zuw. [count] used as with :spellgood.

:spellu[ndo]! {word} Like zuW. [count] used as with :spellgood.

After adding a word to 'spellfile' with the above commands its associated ".spl" file will automatically be updated and reloaded. If you change 'spellfile' manually you need to use the :mkspell command. This sequence of commands mostly works well:

```
:edit <file in 'spellfile'>

        (make changes to the spell file)

:mkspell! %
```

More details about the 'spellfile' format below spell-wordlist-format.

internal-wordlist

The internal word list is used for all buffers where 'spell' is set. It is not stored, it is lost when you exit Vim. It is also cleared when 'encoding' is set.

Finding suggestions for bad words:

z=

z= For the word under/after the cursor suggest correctly spelled words. This also works to find alternatives for a word that is not highlighted as a bad word, e.g., when the word after it is bad.
In Visual mode the highlighted text is taken as the word to be replaced.
The results are sorted on similarity to the word being replaced.
This may take a long time. Hit CTRL-C when you get bored.

If the command is used without a count the alternatives are listed and you can enter the number of your choice or press <Enter> if you don't want to replace. You can also use the mouse to click on your choice (only works if the mouse can be used in Normal mode and when there are no line wraps). Click on the first line (the header) to cancel.

The suggestions listed normally replace a highlighted bad word. Sometimes they include other text, in that case the replaced text is also listed after a "<".

If a count is used that suggestion is used, without prompting. For example, "1z=" always takes the first suggestion.

If 'verbose' is non-zero a score will be displayed
with the suggestions to indicate the likeliness to the
badly spelled word (the higher the score the more
different).
When a word was replaced the redo command "." will
repeat the word replacement. This works like "ciw",
the good word and <Esc>. This does NOT work for Thai
and other languages without spaces between words.

:spellr :spellrepall E752 E753

:spellr[epall] Repeat the replacement done by z= for all matches
with the replaced word in the current window.

In Insert mode, when the cursor is after a badly spelled word, you can use CTRL-X s to find suggestions. This works like Insert mode completion. Use CTRL-N to use the next suggestion, CTRL-P to go back. i_CTRL-X_s

The 'spellsuggest' option influences how the list of suggestions is generated and sorted. See 'spellsuggest'.

The 'spellcapcheck' option is used to check the first word of a sentence starts with a capital. When there is a line break right after a sentence the highlighting of the next line may be postponed. Use CTRL-L when needed. Also see set-spc-auto for how it can be set automatically when 'spelllang' is set.

The 'spelloptions' option has a few more flags that influence the way spell checking works. For example, "camel" splits CamelCased words so that each part of the word is spell-checked separately.

Vim counts the number of times a good word is encountered. This is used to sort the suggestions: words that have been seen before get a small bonus, words that have been seen often get a bigger bonus. The COMMON item in the affix file can be used to define common words, so that this mechanism also works in a new or short file spell-COMMON.

2. Remarks on spell checking spell-remarks

PERFORMANCE

Vim does on-the-fly spell checking. To make this work fast the word list is loaded in memory. Thus this uses a lot of memory (1 Mbyte or more). There might also be a noticeable delay when the word list is loaded, which happens when 'spell' is set and when 'spelllang' is set while 'spell' was already set. To minimize the delay each word list is only loaded once, it is not deleted when 'spelllang' is made empty or 'spell' is reset. When 'encoding' is set all the word lists are reloaded, thus you may notice a delay then too.

REGIONS

A word may be spelled differently in various regions. For example, English comes in (at least) these variants:

en	all regions
en_au	Australia
en_ca	Canada
en_gb	Great Britain
en_nz	New Zealand
en_us	USA

Words that are not used in one region but are used in another region are highlighted with SpellLocal hl-SpellLocal.

Always use lowercase letters for the language and region names.

When adding a word with zg or another command it's always added for all regions. You can change that by manually editing the 'spellfile'. See spell-wordlist-format. Note that the regions as specified in the files in 'spellfile' are only used when all entries in 'spelllang' specify the same region (not counting files specified by their .spl name).

spell-german

Specific exception: For German these special regions are used:

de	all German words accepted
de_de	old and new spelling
de_19	old spelling
de_20	new spelling
de_at	Austria
de_ch	Switzerland

spell-russian

Specific exception: For Russian these special regions are used:

ru	all Russian words accepted
ru_ru	"IE" letter spelling
ru_yo	"YO" letter spelling

spell-yiddish

Yiddish requires using "utf-8" encoding, because of the special characters used. If you are using latin1 Vim will use transliterated (romanized) Yiddish instead. If you want to use transliterated Yiddish with utf-8 use "yi-tr". In a table:

'encoding'	'spelllang'	
utf-8	yi	Yiddish

latin1	yi	transliterated Yiddish
utf-8	yi-tr	transliterated Yiddish

spell-cjk

Chinese, Japanese and other East Asian characters are normally marked as errors, because spell checking of these characters is not supported. If 'spelllang' includes "cjk", these characters are not marked as errors. This is useful when editing text with spell checking while some Asian words are present.

SPELL FILES

spell-load

Vim searches for spell files in the "spell" subdirectory of the directories in 'runtimepath'. The name is: LL.EEE.spl, where:

LL	the language name
EEE	the value of 'encoding'

The value for "LL" comes from 'spelllang', but excludes the region name. Examples:

'spelllang'	LL
en_us	en
en-rare	en-rare
medical_ca	medical

Only the first file is loaded, the one that is first in 'runtimepath'. If this succeeds then additionally files with the name LL.EEE.add.spl are loaded. All the ones that are found are used.

Additionally, the files related to the names in 'spellfile' are loaded. These are the files that zg and zw add good and wrong words to.

Exceptions:

- Vim uses "latin1" when 'encoding' is "iso-8859-15". The euro sign doesn't matter for spelling.
- When no spell file for 'encoding' is found "ascii" is tried. This only works for languages where nearly all words are ASCII, such as English. It helps when 'encoding' is not "latin1", such as iso-8859-2, and English text is being edited. For the ".add" files the same name as the found main spell file is used.

For example, with these values:

```
'runtimepath' is "~/.config/nvim,/usr/share/nvim/runtime/,~/.config/nvim/after"
'encoding'    is "iso-8859-2"
'spelllang'   is "pl"
```

Vim will look for:

1. ~/.config/nvim/spell/pl.iso-8859-2.spl
2. /usr/share/nvim/runtime/spell/pl.iso-8859-2.spl
3. ~/.config/nvim/spell/pl.iso-8859-2.add.spl
4. /usr/share/nvim/runtime/spell/pl.iso-8859-2.add.spl
5. ~/.config/nvim/after/spell/pl.iso-8859-2.add.spl

This assumes 1. is not found and 2. is found.

If 'encoding' is "latin1" Vim will look for:

1. ~/.config/nvim/spell/pl.latin1.spl
2. /usr/share/nvim/runtime/spell/pl.latin1.spl
3. ~/.config/nvim/after/spell/pl.latin1.spl
4. ~/.config/nvim/spell/pl.ascii.spl
5. /usr/share/nvim/runtime/spell/pl.ascii.spl
6. ~/.config/nvim/after/spell/pl.ascii.spl

This assumes none of them are found (Polish doesn't make sense when leaving out the non-ASCII characters).

A spell file might not be available in the current 'encoding'. See spell-mkspell about how to create a spell file. Converting a spell file with "iconv" will NOT work!

spell-sug-file E781

If there is a file with exactly the same name as the ".spl" file but ending in ".sug", that file will be used for giving better suggestions. It isn't loaded before suggestions are made to reduce memory use.

E758 E759 E778 E779 E780 E782 E5042

When loading a spell file Vim checks that it is properly formatted. If you get an error the file may be truncated, modified or intended for another Vim version.

SPELLFILE CLEANUP spellfile-cleanup

The zw command turns existing entries in 'spellfile' into comment lines. This avoids having to write a new file every time, but results in the file only getting longer, never shorter. To clean up the comment lines in all ".add" spell files do this:

```
:runtime spell/cleanadd.vim
```

This deletes all comment lines, except the ones that start with "##". Use "##" lines to add comments that you want to keep.

You can invoke this script as often as you like. A variable is provided to skip updating files that have been changed recently. Set it to the number of seconds that has passed since a file was changed before it will be cleaned. For example, to clean only files that were not changed in the last hour:

```
let g:spell_clean_limit = 60 * 60
```

The default is one second.

WORDS

Vim uses a fixed method to recognize a word. This is independent of 'iskeyword', so that it also works in help files and for languages that include characters like '-' in 'iskeyword'. The word characters do depend on 'encoding'.

The table with word characters is stored in the main .spl file. Therefore it matters what the current locale is when generating it! A .add.spl file does not contain a word table though.

For a word that starts with a digit the digit is ignored, unless the word as a whole is recognized. Thus if "3D" is a word and "D" is not then "3D" is recognized as a word, but if "3D" is not a word then only the "D" is marked as bad. Hex numbers in the form 0x12ab and 0X12AB are recognized.

WORD COMBINATIONS

It is possible to spell-check words that include a space. This is used to recognize words that are invalid when used by themselves, e.g. for "et al.". It can also be used to recognize "the the" and highlight it.

The number of spaces is irrelevant. In most cases a line break may also appear. However, this makes it difficult to find out where to start checking for spelling mistakes. When you make a change to one line and only that line is redrawn Vim won't look in the previous line, thus when "et" is at the end of the previous line "al." will be flagged as an error. And when you type "the<CR>the" the highlighting doesn't appear until the first line is redrawn. Use CTRL-L to redraw right away. "[s" will also stop at a word combination with a line break.

When encountering a line break Vim skips characters such as "*", '>' and '"', so that comments in C, shell and Vim code can be spell checked.

SYNTAX HIGHLIGHTING spell-syntax

Files that use syntax highlighting can specify where spell checking should be done:

1.	everywhere	default
2.	in specific items	use "contains=@Spell"
3.	everywhere but specific items	use "contains=@NoSpell"

For the second method adding the @NoSpell cluster will disable spell checking again. This can be used, for example, to add @Spell to the comments of a program, and add @NoSpell for items that shouldn't be checked.
Also see :syn-spell for text that is not in a syntax item.

VIM SCRIPTS

If you want to write a Vim script that does something with spelling, you may find these functions useful:

spellbadword()	find badly spelled word at the cursor
spellsuggest()	get list of spelling suggestions
soundfold()	get the sound-a-like version of a word

SETTING 'spellcapcheck' AUTOMATICALLY set-spc-auto

After the 'spelllang' option has been set successfully, Vim will source the files "spell/LANG.vim" and "spell/LANG.lua" in 'runtimepath'. "LANG" is the value of 'spelllang' up to the first comma, dot or underscore. This can be used to set options specifically for the language, especially 'spellcapcheck'.

The distribution includes a few of these files. Use this command to see what they do:

```
:next $VIMRUNTIME/spell/*.vim
```

Note that the default scripts don't set 'spellcapcheck' if it was changed from the default value. This assumes the user prefers another value then.

DOUBLE SCORING spell-double-scoring

The 'spellsuggest' option can be used to select "double" scoring. This mechanism is based on the principle that there are two kinds of spelling mistakes:

1. You know how to spell the word, but mistype something. This results in a small editing distance (character swapped/omitted/inserted) and possibly a word that sounds completely different.
2. You don't know how to spell the word and type something that sounds right. The edit distance can be big but the word is similar after sound-folding.

Since scores for these two mistakes will be very different we use a list for each and mix them.

The sound-folding is slow and people that know the language won't make the second kind of mistakes. Therefore 'spellsuggest' can be set to select the preferred method for scoring the suggestions.

3. Generating a spell file spell-mkspell

Vim uses a binary file format for spelling. This greatly speeds up loading the word list and keeps it small.

.aff .dic Myspell

You can create a Vim spell file from the .aff and .dic files that Myspell uses. Myspell is used by OpenOffice.org and Mozilla. The OpenOffice .oxt files are zip files which contain the .aff and .dic files. You should be able to find them here:

https://extensions.openoffice.org/en/search@f%5B0%5D%3Dfield_project_tags%253A311.html

The older, OpenOffice 2 files may be used if this doesn't work:

http://wiki.services.openoffice.org/wiki/Dictionaries

You can also use a plain word list. The results are the same, the choice depends on what word lists you can find.

Make sure your current locale is set properly, otherwise Vim doesn't know what characters are upper/lower case letters. If the locale isn't available (e.g., when using an MS-Windows codepage on Unix) add tables to the .aff file spell-affix-chars. If the .aff file doesn't define a table then the word table of the currently active spelling is used. If spelling is not active then Vim will try to guess.

:mksp :mkspell

:mksp[ell][!] [-ascii] {outname} {inname} ...
Generate a Vim spell file from word lists. Example:

```
:mkspell /tmp/nl nl_NL.words
```

E751

When {outname} ends in ".spl" it is used as the output file name. Otherwise it should be a language name, such as "en", without the region name. The file written will be "{outname}.{encoding}.spl", where {encoding} is the value of the 'encoding' option.

When the output file already exists [!] must be used to overwrite it.

When the [-ascii] argument is present, words with non-ascii characters are skipped. The resulting file ends in "ascii.spl".

The input can be the Myspell format files {inname}.aff and {inname}.dic. If {inname}.aff does not exist then {inname} is used as the file name of a plain word list.

Multiple {inname} arguments can be given to combine regions into one Vim spell file. Example:

```
:mkspell ~/.config/nvim/spell/en /tmp/en_US /tmp/en_CA /tmp/en_AU
```

This combines the English word lists for US, CA and AU into one en.spl file.
Up to eight regions can be combined. E754 E755
The REP and SAL items of the first .aff file where

they appear are used. spell-REP spell-SAL

E845

This command uses a lot of memory, required to find the optimal word tree (Polish, Italian and Hungarian require several hundred Mbyte). The final result will be much smaller, because compression is used. To avoid running out of memory compression will be done now and then. This can be tuned with the 'mkspellmem' option.

E1578

There is a limit on how many postponed prefix and compound flags can be stored for one word. Reduce the number of affix/compound flags on a word in the .dic file that exceeds it.

After the spell file was written and it was being used in a buffer it will be reloaded automatically.

:mksp[ell] [-ascii] {name}.{enc}.add
Like ":mkspell" above, using {name}.{enc}.add as the input file and producing an output file in the same directory that has ".spl" appended.

:mksp[ell] [-ascii] {name}
Like ":mkspell" above, using {name} as the input file and producing an output file in the same directory that has ".{enc}.spl" appended.

Vim will report the number of duplicate words. This might be a mistake in the list of words. But sometimes it is used to have different prefixes and suffixes for the same basic word to avoid them combining (e.g. Czech uses this). If you want Vim to report all duplicate words set the 'verbose' option.

Since you might want to change a Myspell word list for use with Vim the following procedure is recommended:

1. Obtain the xx_YY.aff and xx_YY.dic files from Myspell.

2. Make a copy of these files to xx_YY.orig.aff and xx_YY.orig.dic.

3. Change the xx_YY.aff and xx_YY.dic files to remove bad words, add missing words, define word characters with FOL/LOW/UPP, etc. The distributed "*.diff" files can be used.

4. Start Vim with the right locale and use :mkspell to generate the Vim spell file.

5. Try out the spell file with ":set spell spelllang=xx" if you wrote it in a spell directory in 'runtimepath', or ":set spelllang=xx.enc.spl" if you wrote it somewhere else.

When the Myspell files are updated you can merge the differences:

1. Obtain the new Myspell files as xx_YY.new.aff and xx_UU.new.dic.

2. Use diff-mode to see what changed:

```
nvim -d xx_YY.orig.dic xx_YY.new.dic
```

3. Take over the changes you like in xx_YY.dic. You may also need to change xx_YY.aff.

4. Rename xx_YY.new.dic to xx_YY.orig.dic and xx_YY.new.aff to xx_YY.orig.aff.

SPELL FILE VERSIONS

E770 E771 E772

Spell checking is a relatively new feature in Vim, thus it's possible that the .spl file format will be changed to support more languages. Vim will check the validity of the spell file and report anything wrong.

E771: Old spell file, needs to be updated
This spell file is older than your Vim. You need to update the .spl file.

E772: Spell file is for newer version of Vim
This means the spell file was made for a later version of Vim. You need to update Vim.

E770: Unsupported section in spell file
This means the spell file was made for a later version of Vim and contains a section that is required for the spell file to work. In this case it's probably a good idea to upgrade your Vim.

SPELL FILE DUMP

If for some reason you want to check what words are supported by the currently used spelling files, use this command:

:spelldump :spelld

:spelld[ump] Open a new window and fill it with all currently valid words. Compound words are not included.
Note: For some languages the result may be enormous, causing Vim to run out of memory.

:spelld[ump]! Like ":spelldump" and include the word count. This is the number of times the word was found while updating the screen. Words that are in COMMON items get a starting count of 10.

The format of the word list is used spell-wordlist-format. You should be able to read it with ":mkspell" to generate one .spl file that includes all the words.

When all entries to 'spelllang' use the same regions or no regions at all then the region information is included in the dumped words. Otherwise only words for the current region are included and no "/regions" line is generated.

Comment lines with the name of the .spl file are used as a header above the words that were generated from that .spl file.

SPELL FILE MISSING spell-SpellFileMissing

If a spell file is missing, the user is asked whether to download it. See package-spellfile.

E797

Note that the SpellFileMissing autocommand must not change or destroy the buffer the user was editing.

4. Spell file format spell-file-format

This is the format of the files that are used by the person who creates and maintains a word list.

Note that we avoid the word "dictionary" here. That is because the goal of spell checking differs from writing a dictionary (as in the book). For spelling we need a list of words that are OK, thus should not be highlighted. Person and company names will not appear in a dictionary, but do appear in a word list. And some old words are rarely used while they are common misspellings. These do appear in a dictionary but not in a word list.

There are two formats: A straight list of words and a list using affix compression. The files with affix compression are used by Myspell (Mozilla and OpenOffice.org). This requires two files, one with .aff and one with .dic extension.

FORMAT OF STRAIGHT WORD LIST spell-wordlist-format

The words must appear one per line. That is all that is required.

Additionally the following items are recognized:

- Empty and blank lines are ignored.

comment
- Lines starting with a # are ignored (comment lines).

/encoding=utf-8
- A line starting with "/encoding=", before any word, specifies the encoding

  of the file. After the second '=' comes an encoding name. This tells Vim to setup conversion from the specified encoding to 'encoding'. Thus you can use one word list for several target encodings.

/regions=usca
- A line starting with "/regions=" specifies the region names that are

  supported. Each region name must be two ASCII letters. The first one is region 1. Thus "/regions=usca" has region 1 "us" and region 2 "ca". In an addition word list the region names should be equal to the main word list!
- Other lines starting with '/' are reserved for future use. The ones that

  are not recognized are ignored. You do get a warning message, so that you know something won't work.
- A "/" may follow the word with the following items:

  = Case must match exactly.
  ? Rare word.
  ! Bad (wrong) word.

1 to 9 A region in which the word is valid. If no regions are specified the word is valid in all regions.

Example:

```
# This is an example word list          comment
/encoding=latin1                        encoding of the file
/regions=uscagb                         regions "us", "ca" and "gb"
example                                 word for all regions
blah/12                                 word for regions "us" and "ca"
vim/!                                   bad word
Campbell/?3                             rare word in region 3 "gb"
's mornings/=                           keep-case word
```

Note that when "/=" is used the same word with all upper-case letters is not accepted. This is different from a word with mixed case that is automatically marked as keep-case, those words may appear in all upper-case letters.

FORMAT WITH .AFF AND .DIC FILES aff-dic-format

There are two files: the basic word list and an affix file. The affix file specifies settings for the language and can contain affixes. The affixes are used to modify the basic words to get the full word list. This significantly reduces the number of words, especially for a language like Polish. This is called affix compression.

The basic word list and the affix file are combined with the ":mkspell" command and results in a binary spell file. All the preprocessing has been done, thus this file loads fast. The binary spell file format is described in the source code (src/spell.c). But only developers need to know about it.

The preprocessing also allows us to take the Myspell language files and modify them before the Vim word list is made. The tools for this can be found in the "src/spell" directory.

The format for the affix and word list files is based on what Myspell uses (the spell checker of Mozilla and OpenOffice.org). A description can be found here:

https://lingucomponent.openoffice.org/affix.readme

Note that affixes are case sensitive, this isn't obvious from the description.

Vim supports quite a few extras. They are described below spell-affix-vim. Attempts have been made to keep this compatible with other spell checkers, so that the same files can often be used. One other project that offers more than Myspell is Hunspell (https://hunspell.github.io).

WORD LIST FORMAT spell-dic-format

A short example, with line numbers:

```
1                                       1234
2 aan
3 Als
4 Etten-Leur
5 et al.
6 's-Gravenhage
7 's-Gravenhaags
8 # word that differs between regions
9 kado/1
10 cadeau/2
11 TCP,IP
12 /the S affix may add a 's'
13 bedel/S
```

The first line contains the number of words. Vim ignores it, but you do get an error message if it's not there. E760

What follows is one word per line. White space at the end of the line is ignored, all other white space matters. The encoding is specified in the affix file spell-SET.

Comment lines start with '#' or '/'. See the example lines 8 and 12. Note that putting a comment after a word is NOT allowed:

```
someword   # comment that causes an error!
```

After the word there is an optional slash and flags. Most of these flags are letters that indicate the affixes that can be used with this word. These are specified with SFX and PFX lines in the .aff file, see spell-SFX and spell-PFX. Vim allows using other flag types with the FLAG item in the affix file spell-FLAG.

When the word only has lower-case letters it will also match with the word starting with an upper-case letter.

When the word includes an upper-case letter, this means the upper-case letter is required at this position. The same word with a lower-case letter at this position will not match. When some of the other letters are upper-case it will not match either.

The word with all upper-case characters will always be OK,

word list	matches	does not match
als	als Als ALS	ALs AlS aLs aLS
Als	Als ALS	als ALs AlS aLs aLS
ALS	ALS	als Als ALs AlS aLs aLS
AlS	AlS ALS	als Als ALs aLs aLS

The KEEPCASE affix ID can be used to specifically match a word with identical case only, see below spell-KEEPCASE.

Note: in line 5 to 7 non-word characters are used. You can include any character in a word. When checking the text a word still only matches when it appears with a non-word character before and after it. For Myspell a word starting with a non-word character probably won't work.

In line 12 the word "TCP/IP" is defined. Since the slash has a special meaning the comma is used instead. This is defined with the SLASH item in the affix file, see spell-SLASH. Note that without this SLASH item the word will be "TCP,IP".

AFFIX FILE FORMAT

spell-aff-format spell-affix-vim

spell-affix-comment

Comment lines in the .aff file start with a '#':

```
# comment line
```

Items with a fixed number of arguments can be followed by a comment. But only if none of the arguments can contain white space. The comment must start with a "#" character. Example:

```
KEEPCASE =  # fix case for words with this flag
```

ENCODING

spell-SET

The affix file can be in any encoding that is supported by "iconv". However, in some cases the current locale should also be set properly at the time :mkspell is invoked. Adding FOL/LOW/UPP lines removes this requirement spell-FOL.

The encoding should be specified before anything where the encoding matters. The encoding applies both to the affix file and the dictionary file. It is done with a SET line:

```
SET utf-8
```

The encoding can be different from the value of the 'encoding' option at the time ":mkspell" is used. Vim will then convert everything to 'encoding' and generate a spell file for 'encoding'. If some of the used characters to not fit in 'encoding' you will get an error message.

spell-affix-mbyte

When using a multibyte encoding it's possible to use more different affix flags. But Myspell doesn't support that, thus you may not want to use it anyway. For compatibility use an 8-bit encoding.

INFORMATION

These entries in the affix file can be used to add information to the spell file. There are no restrictions on the format, but they should be in the right encoding.

spell-NAME spell-VERSION spell-HOME
spell-AUTHOR spell-EMAIL spell-COPYRIGHT

```
NAME            Name of the language
VERSION         1.0.1  with fixes
HOME            https://www.example.com
AUTHOR          John Doe
EMAIL           john AT Doe DOT net
COPYRIGHT       LGPL
```

These fields are put in the .spl file as-is. The :spellinfo command can be used to view the info.

:spellinfo :spelli

```
:spelli[nfo]            Display the information for the spell file(s) used for
                        the current buffer.
```

CHARACTER TABLES

spell-affix-chars

When using an 8-bit encoding the affix file should define what characters are word characters. This is because the system where ":mkspell" is used may not support a locale with this encoding and isalpha() won't work. For example when using "cp1250" on Unix.

E761 E762 spell-FOL
spell-LOW spell-UPP

Three lines in the affix file are needed. Simplistic example:

```
FOL  áëñ
LOW  áëñ
UPP  ÁËÑ
```

All three lines must have exactly the same number of characters.

The "FOL" line specifies the case-folded characters. These are used to compare words while ignoring case. For most encodings this is identical to the lower case line.

The "LOW" line specifies the characters in lower-case. Mostly it's equal to the "FOL" line.

The "UPP" line specifies the characters with upper-case. That is, a character is upper-case where it's different from the character at the same position in "FOL".

An exception is made for the German sharp s ß. The upper-case version is "SS". In the FOL/LOW/UPP lines it should be included, so that it's recognized as a word character, but use the ß character in all three.

ASCII characters should be omitted, Vim always handles these in the same way. When the encoding is UTF-8 no word characters need to be specified.

E763

Vim allows you to use spell checking for several languages in the same file. You can list them in the 'spelllang' option. As a consequence all spell files for the same encoding must use the same word characters, otherwise they can't be combined without errors.

If you get an E763 warning that the word tables differ you need to update your ".spl" spell files. If you downloaded the files, get the latest version of all spell files you use. If you are only using one, e.g., German, then also download the recent English spell files. Otherwise generate the .spl file again with :mkspell. If you still get errors check the FOL, LOW and UPP lines in the used .aff files.

The XX.ascii.spl spell file generated with the "-ascii" argument will not contain the table with characters, so that it can be combine with spell files for any encoding. The .add.spl files also do not contain the table.

MID-WORD CHARACTERS

spell-midword

Some characters are only to be considered word characters if they are used in between two ordinary word characters. An example is the single quote: It is often used to put text in quotes, thus it can't be recognized as a word character, but when it appears in between word characters it must be part of the word. This is needed to detect a spelling error such as they'are. That should be they're, but since "they" and "are" are words themselves that would go unnoticed.

These characters are defined with MIDWORD in the .aff file. Example:

```
MIDWORD '-
```

FLAG TYPES

spell-FLAG

Flags are used to specify the affixes that can be used with a word and for other properties of the word. Normally single-character flags are used. This limits the number of possible flags, especially for 8-bit encodings. The FLAG item can be used if more affixes are to be used. Possible values:

```
FLAG long       use two-character flags
FLAG num        use numbers, from 1 up to 65000
FLAG caplong    use one-character flags without A-Z and two-character
                flags that start with A-Z
```

With "FLAG num" the numbers in a list of affixes need to be separated with a comma: "234,2143,1435". This method is inefficient, but useful if the file is generated with a program.

When using "caplong" the two-character flags all start with a capital: "Aa", "B1", "BB", etc. This is useful to use one-character flags for the most common items and two-character flags for uncommon items.

Note: When using utf-8 only characters up to 65000 may be used for flags.

Note: even when using "num" or "long" the number of flags available to compounding and prefixes is limited to about 250.

AFFIXES

spell-PFX spell-SFX

The usual PFX (prefix) and SFX (suffix) lines are supported (see the Myspell documentation or the Aspell manual: http://aspell.net/man-html/Affix-Compression.html).

Summary:

```
SFX L Y 2
SFX L 0 re [^x]
SFX L 0 ro x
```

The first line is a header and has four fields:

```
SFX {flag} {combine} {count}
```

{flag} The name used for the suffix. Mostly it's a single letter, but other characters can be used, see spell-FLAG.

{combine} Can be 'Y' or 'N'. When 'Y' then the word plus suffix can also have a prefix. When 'N' then a prefix is not allowed.

{count} The number of lines following. If this is wrong you will get an error message.

For PFX the fields are exactly the same.

The basic format for the following lines is:

```
SFX {flag} {strip} {add} {condition} {extra}
```

{flag} Must be the same as the {flag} used in the first line.

{strip} Characters removed from the basic word. There is no check if the characters are actually there, only the length is used (in bytes). This better match the {condition}, otherwise strange things may happen. If the {strip} length is equal to or longer than the basic word the suffix won't be used. When {strip} is 0 (zero) then nothing is stripped.

{add} Characters added to the basic word, after removing {strip}. Optionally there is a '/' followed by flags. The flags apply to the word plus affix. See spell-affix-flags

{condition} A simplistic pattern. Only when this matches with a basic word will the suffix be used for that word. This is normally for using one suffix letter with different {add} and {strip} fields for words with different endings. When {condition} is a . (dot) there is no condition. The pattern may contain:

- Literal characters.
- A set of characters in []. [abc] matches a, b and c. A dash is allowed for a range [a-c], but this is Vim-specific.
- A set of characters that starts with a ^, meaning the complement of the specified characters. [^abc] matches any character but a, b and c.

{extra} Optional extra text:

```
# comment           Comment is ignored
-                   Hunspell uses this, ignored
```

For PFX the fields are the same, but the {strip}, {add} and {condition} apply to the start of the word.

Note: Myspell ignores any extra text after the relevant info. Vim requires this text to start with a "#" so that mistakes don't go unnoticed. Example:

```
SFX F 0 in   [^i]n      # Spion > Spionin
SFX F 0 nen  in         # Bauerin > Bauerinnen
```

However, to avoid lots of errors in affix files written for Myspell, you can add the IGNOREEXTRA flag.

Apparently Myspell allows an affix name to appear more than once. Since this might also be a mistake, Vim checks for an extra "S". The affix files for Myspell that use this feature apparently have this flag. Example:

```
SFX a Y 1 S
SFX a 0 an .

SFX a Y 2 S
SFX a 0 en .
SFX a 0 on .
```

AFFIX FLAGS spell-affix-flags

This is a feature that comes from Hunspell: The affix may specify flags. This works similar to flags specified on a basic word. The flags apply to the basic word plus the affix (but there are restrictions). Example:

```
SFX S Y 1
SFX S 0 s .

SFX A Y 1
SFX A 0 able/S .
```

When the dictionary file contains "drink/AS" then these words are possible:

```
drink
drinks          uses S suffix
drinkable       uses A suffix
drinkables      uses A suffix and then S suffix
```

Generally the flags of the suffix are added to the flags of the basic word, both are used for the word plus suffix. But the flags of the basic word are only used once for affixes, except that both one prefix and one suffix can be used when both support combining.

Specifically, the affix flags can be used for:

- Suffixes on suffixes, as in the example above. This works once, thus you can have two suffixes on a word (plus one prefix).
- Making the word with the affix rare, by using the spell-RARE flag.
- Exclude the word with the affix from compounding, by using the spell-COMPOUNDFORBIDFLAG flag.

- Allow the word with the affix to be part of a compound word on the side of the affix with the spell-COMPOUNDPERMITFLAG.
- Use the NEEDCOMPOUND flag: word plus affix can only be used as part of a compound word. spell-NEEDCOMPOUND
- Compound flags: word plus affix can be part of a compound word at the end, middle, start, etc. The flags are combined with the flags of the basic word. spell-compound
- NEEDAFFIX: another affix is needed to make a valid word.

- CIRCUMFIX, as explained just below.

IGNOREEXTRA spell-IGNOREEXTRA

Normally Vim gives an error for an extra field that does not start with '#'. This avoids errors going unnoticed. However, some files created for Myspell or Hunspell may contain many entries with an extra field. Use the IGNOREEXTRA flag to avoid lots of errors.

CIRCUMFIX spell-CIRCUMFIX

The CIRCUMFIX flag means a prefix and suffix must be added at the same time. If a prefix has the CIRCUMFIX flag then only suffixes with the CIRCUMFIX flag can be added, and the other way around.
An alternative is to only specify the suffix, and give that suffix two flags: the required prefix and the NEEDAFFIX flag. spell-NEEDAFFIX

PFXPOSTPONE spell-PFXPOSTPONE

When an affix file has very many prefixes that apply to many words it's not possible to build the whole word list in memory. This applies to Hebrew (a list with all words is over a Gbyte). In that case applying prefixes must be postponed. This makes spell checking slower. It is indicated by this keyword in the .aff file:

PFXPOSTPONE

Only prefixes without a chop string and without flags can be postponed. Prefixes with a chop string or with flags will still be included in the word list. An exception if the chop string is one character and equal to the last character of the added string, but in lower case. Thus when the chop string is used to allow the following word to start with an upper case letter.

WORDS WITH A SLASH spell-SLASH

The slash is used in the .dic file to separate the basic word from the affix letters and other flags. Unfortunately, this means you cannot use a slash in a word. Thus "TCP/IP" is not a word but "TCP" with the flags "IP". To include a slash in the word put a backslash before it: "TCP\/IP". In the rare case you want to use a backslash inside a word you need to use two backslashes.
Any other use of the backslash is reserved for future expansion.

KEEP-CASE WORDS spell-KEEPCASE

In the affix file a KEEPCASE line can be used to define the affix name used for keep-case words. Example:

KEEPCASE =

This flag is not supported by Myspell. It has the meaning that case matters. This can be used if the word does not have the first letter in upper case at the start of a sentence. Example:

word list	matches	does not match
's morgens/= 's morgens	'S morgens 's Morgens 'S MORGENS	
's Morgens	's Morgens 'S MORGENS 'S morgens 's morgens	

The flag can also be used to avoid that the word matches when it is in all upper-case letters.

RARE WORDS spell-RARE

In the affix file a RARE line can be used to define the affix name used for rare words. Example:

RARE ?

Rare words are highlighted differently from bad words. This is to be used for words that are correct for the language, but are hardly ever used and could be a typing mistake anyway.

This flag can also be used on an affix, so that a basic word is not rare but the basic word plus affix is rare spell-affix-flags. However, if the word also appears as a good word in another way (e.g., in another region) it won't be marked as rare.

BAD WORDS

spell-BAD

In the affix file a BAD line can be used to define the affix name used for bad words. Example:

BAD !

This can be used to exclude words that would otherwise be good. For example "the the" in the .dic file:

the the/!

Once a word has been marked as bad it won't be undone by encountering the same word as good.

The flag also applies to the word with affixes, thus this can be used to mark a whole bunch of related words as bad.

spell-FORBIDDENWORD

FORBIDDENWORD can be used just like BAD. For compatibility with Hunspell.

spell-NEEDAFFIX

The NEEDAFFIX flag is used to require that a word is used with an affix. The word itself is not a good word (unless there is an empty affix). Example:

NEEDAFFIX +

COMPOUND WORDS

spell-compound

A compound word is a longer word made by concatenating words that appear in the .dic file. To specify which words may be concatenated a character is used. This character is put in the list of affixes after the word. We will call this character a flag here. Obviously these flags must be different from any affix IDs used.

spell-COMPOUNDFLAG

The Myspell compatible method uses one flag, specified with COMPOUNDFLAG. All words with this flag combine in any order. This means there is no control over which word comes first. Example:
COMPOUNDFLAG c

spell-COMPOUNDRULE

A more advanced method to specify how compound words can be formed uses multiple items with multiple flags. This is not compatible with Myspell 3.0. Let's start with an example:
COMPOUNDRULE c+
COMPOUNDRULE se

The first line defines that words with the "c" flag can be concatenated in any order. The second line defines compound words that are made of one word with the "s" flag and one word with the "e" flag. With this dictionary:
bork/c
onion/s
soup/e

You can make these words:
```
        bork
        borkbork
        borkborkbork
        (etc.)
        onion
        soup
        onionsoup
```

The COMPOUNDRULE item may appear multiple times. The argument is made out of one or more groups, where each group can be:
```
        one flag                        e.g., c
        alternate flags inside []       e.g., [abc]
```
Optionally this may be followed by:
```
        *       the group appears zero or more times, e.g., sm*e
        +       the group appears one or more times, e.g., c+
        ?       the group appears zero times or once, e.g., x?
```

This is similar to the regexp pattern syntax (but not the same!). A few examples with the sequence of word flags they require:
```
    COMPOUNDRULE x+         x xx xxx etc.
    COMPOUNDRULE yz         yz
    COMPOUNDRULE x+z        xz xxz xxxz etc.
    COMPOUNDRULE yx+        yx yxx yxxx etc.
    COMPOUNDRULE xy?z       xz xyz

    COMPOUNDRULE [abc]z    az bz cz
    COMPOUNDRULE [abc]+z   az aaz abaz bz baz bcbz cz caz cbaz etc.
    COMPOUNDRULE a[xyz]+   ax axx axyz ay ayx ayzz az azy azxy etc.
    COMPOUNDRULE sm*e       se sme smme smmme etc.
    COMPOUNDRULE s[xyz]*e  se sxe sxye sxyxe sye syze sze szye szyxe  etc.
```

A specific example: Allow a compound to be made of two words and a dash:
        In the .aff file:
COMPOUNDRULE sde
NEEDAFFIX x
COMPOUNDWORDMAX 3
COMPOUNDMIN 1
        In the .dic file:
start/s

end/e
-/xd

This allows for the word "start-end", but not "startend".

An additional implied rule is that, without further flags, a word with a prefix cannot be compounded after another word, and a word with a suffix cannot be compounded with a following word. Thus the affix cannot appear on the inside of a compound word. This can be changed with the spell-COMPOUNDPERMITFLAG.

spell-NEEDCOMPOUND

The NEEDCOMPOUND flag is used to require that a word is used as part of a compound word. The word itself is not a good word. Example:

NEEDCOMPOUND &

spell-ONLYINCOMPOUND

The ONLYINCOMPOUND does exactly the same as NEEDCOMPOUND. Supported for compatibility with Hunspell.

spell-COMPOUNDMIN

The minimal character length of a word used for compounding is specified with COMPOUNDMIN. Example:
COMPOUNDMIN 5

When omitted there is no minimal length. Obviously you could just leave out the compound flag from short words instead, this feature is present for compatibility with Myspell.

spell-COMPOUNDWORDMAX

The maximum number of words that can be concatenated into a compound word is specified with COMPOUNDWORDMAX. Example:
COMPOUNDWORDMAX 3

When omitted there is no maximum. It applies to all compound words.

To set a limit for words with specific flags make sure the items in COMPOUNDRULE where they appear don't allow too many words.

spell-COMPOUNDSYLMAX

The maximum number of syllables that a compound word may contain is specified with COMPOUNDSYLMAX. Example:
COMPOUNDSYLMAX 6

This has no effect if there is no SYLLABLE item. Without COMPOUNDSYLMAX there is no limit on the number of syllables.

If both COMPOUNDWORDMAX and COMPOUNDSYLMAX are defined, a compound word is accepted if it fits one of the criteria, thus is either made from up to COMPOUNDWORDMAX words or contains up to COMPOUNDSYLMAX syllables.

spell-COMPOUNDFORBIDFLAG

The COMPOUNDFORBIDFLAG specifies a flag that can be used on an affix. It means that the word plus affix cannot be used in a compound word. Example:
affix file:
COMPOUNDFLAG c
COMPOUNDFORBIDFLAG x
SFX a Y 2
SFX a 0 s .
SFX a 0 ize/x .
dictionary:
word/c
util/ac

This allows for "wordutil" and "wordutils" but not "wordutilize".
Note: this doesn't work for postponed prefixes yet.

spell-COMPOUNDPERMITFLAG

The COMPOUNDPERMITFLAG specifies a flag that can be used on an affix. It means that the word plus affix can also be used in a compound word in a way where the affix ends up halfway through the word. Without this flag that is not allowed.
Note: this doesn't work for postponed prefixes yet.

spell-COMPOUNDROOT

The COMPOUNDROOT flag is used for words in the dictionary that are already a compound. This means it counts for two words when checking the compounding rules. Can also be used for an affix to count the affix as a compounding word.

spell-CHECKCOMPOUNDPATTERN

CHECKCOMPOUNDPATTERN is used to define patterns that, when matching at the position where two words are compounded together forbids the compound.
For example:
CHECKCOMPOUNDPATTERN o e

This forbids compounding if the first word ends in "o" and the second word starts with "e".

The arguments must be plain text, no patterns are actually supported, despite the item name. Case is always ignored.

The Hunspell feature to use three arguments and flags is not supported.

spell-NOCOMPOUNDSUGS

This item indicates that using compounding to make suggestions is not a good idea. Use this when compounding is used with very short or one-character words. E.g. to make numbers out of digits. Without this flag creating suggestions would spend most time trying all kind of weird compound words.

NOCOMPOUNDSUGS

spell-SYLLABLE

The SYLLABLE item defines characters or character sequences that are used to count the number of syllables in a word. Example:
SYLLABLE aáeéiíoóöõuúüûy/aa/au/ea/ee/ei/ie/oa/oe/oo/ou/uu/ui

Before the first slash is the set of characters that are counted for one syllable, also when repeated and mixed, until the next character that is not in this set. After the slash come sequences of characters that are counted for one syllable. These are preferred over using characters from the set. With the example "ideeen" has three syllables, counted by "i", "ee" and "e".

Only case-folded letters need to be included.

Another way to restrict compounding was mentioned above: Adding the spell-COMPOUNDFORBIDFLAG flag to an affix causes all words that are made with that affix to not be used for compounding.

UNLIMITED COMPOUNDING

spell-NOBREAK

For some languages, such as Thai, there is no space in between words. This looks like all words are compounded. To specify this use the NOBREAK item in the affix file, without arguments:
NOBREAK

Vim will try to figure out where one word ends and a next starts. When there are spelling mistakes this may not be quite right.

spell-COMMON

Common words can be specified with the COMMON item. This will give better suggestions when editing a short file. Example:

COMMON the of to and a in is it you that he she was for on are

The words must be separated by white space, up to 25 per line.
When multiple regions are specified in a ":mkspell" command the common words for all regions are combined and used for all regions.

spell-NOSPLITSUGS

This item indicates that splitting a word to make suggestions is not a good idea. Split-word suggestions will appear only when there are few similar words.

NOSPLITSUGS

spell-NOSUGGEST

The flag specified with NOSUGGEST can be used for words that will not be suggested. Can be used for obscene words.

NOSUGGEST %

REPLACEMENTS

spell-REP

In the affix file REP items can be used to define common mistakes. This is used to make spelling suggestions. The items define the "from" text and the "to" replacement. Example:

```
REP 4
REP f ph
REP ph f
REP k ch
REP ch k
```

The first line specifies the number of REP lines following. Vim ignores the number, but it must be there (for compatibility with Myspell).

Don't include simple one-character replacements or swaps. Vim will try these anyway. You can include whole words if you want to, but you might want to use the "file:" item in 'spellsuggest' instead.

You can include a space by using an underscore:

REP the_the the

SIMILAR CHARACTERS

spell-MAP **E783**

In the affix file MAP items can be used to define letters that are very much alike. This is mostly used for a letter with different accents. This is used to prefer suggestions with these letters substituted. Example:

```
MAP 2
MAP eéëêè
MAP uüùúû
```

The first line specifies the number of MAP lines following. Vim ignores the number, but the line must be there.

Each letter must appear in only one of the MAP items. It's a bit more efficient if the first letter is ASCII or at least one without accents.

.SUG FILE

spell-NOSUGFILE

When soundfolding is specified in the affix file then ":mkspell" will normally produce a .sug file next to the .spl file. This file is used to find suggestions by their sound-a-like form quickly. At the cost of a lot of memory (the amount depends on the number of words, :mkspell will display an estimate when it's done).

To avoid producing a .sug file use this item in the affix file:

NOSUGFILE

Users can simply omit the .sug file if they don't want to use it.

SOUND-A-LIKE

spell-SAL

In the affix file SAL items can be used to define the sounds-a-like mechanism to be used. The main items define the "from" text and the "to" replacement. Simplistic example:

```
SAL CIA      X

SAL CH X
SAL C K
SAL K K
```

There are a few rules and this can become quite complicated. An explanation how it works can be found in the Aspell manual:
http://aspell.net/man-html/Phonetic-Code.html.

There are a few special items:

```
SAL followup                 true

SAL collapse_result true
SAL remove_accents true
```

"1" has the same meaning as "true". Any other value means "false".

SIMPLE SOUNDFOLDING

spell-SOFOFROM spell-SOFOTO

The SAL mechanism is complex and slow. A simpler mechanism is mapping all characters to another character, mapping similar sounding characters to the same character. At the same time this does case folding. You can not have both SAL items and simple soundfolding.

There are two items required: one to specify the characters that are mapped and one that specifies the characters they are mapped to. They must have exactly the same number of characters. Example:

```
SOFOFROM abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
SOFOTO   ebctefghejklnnepkrstevvkesebctefghejklnnepkrstevvkes
```

In the example all vowels are mapped to the same character 'e'. Another method would be to leave out all vowels. Some characters that sound nearly the same and are often mixed up, such as 'm' and 'n', are mapped to the same character. Don't do this too much, all words will start looking alike.

Characters that do not appear in SOFOFROM will be left out, except that all white space is replaced by one space. Sequences of the same character in SOFOFROM are replaced by one.

You can use the soundfold() function to try out the results. Or set the 'verbose' option to see the score in the output of the z= command.

UNSUPPORTED ITEMS

spell-affix-not-supported

These items appear in the affix file of other spell checkers. In Vim they are ignored, not supported or defined in another way.

ACCENT (Hunspell) spell-ACCENT

Use MAP instead. spell-MAP

BREAK (Hunspell) spell-BREAK

Define break points. Unclear how it works exactly.
Not supported.

CHECKCOMPOUNDCASE (Hunspell) spell-CHECKCOMPOUNDCASE

Disallow uppercase letters at compound word boundaries.
Not supported.

CHECKCOMPOUNDDUP (Hunspell) spell-CHECKCOMPOUNDDUP

Disallow using the same word twice in a compound. Not supported.

CHECKCOMPOUNDREP (Hunspell) spell-CHECKCOMPOUNDREP

Something about using REP items and compound words. Not supported.

CHECKCOMPOUNDTRIPLE (Hunspell) spell-CHECKCOMPOUNDTRIPLE

Forbid three identical characters when compounding. Not supported.

CHECKSHARPS (Hunspell) spell-CHECKSHARPS

SS letter pair in uppercased (German) words may be upper case
sharp s (ß). Not supported.

COMPLEXPREFIXES (Hunspell) spell-COMPLEXPREFIXES

Enables using two prefixes. Not supported.

COMPOUND (Hunspell) spell-COMPOUND

This is one line with the count of COMPOUND items, followed by
that many COMPOUND lines with a pattern.
Remove the first line with the count and rename the other
items to COMPOUNDRULE spell-COMPOUNDRULE

COMPOUNDFIRST (Hunspell) spell-COMPOUNDFIRST

Use COMPOUNDRULE instead. spell-COMPOUNDRULE

COMPOUNDBEGIN (Hunspell) spell-COMPOUNDBEGIN

Words signed with COMPOUNDBEGIN may be first elements in
compound words.
Use COMPOUNDRULE instead. spell-COMPOUNDRULE

COMPOUNDLAST (Hunspell) spell-COMPOUNDLAST

Words signed with COMPOUNDLAST may be last elements in
compound words.
Use COMPOUNDRULE instead. spell-COMPOUNDRULE

COMPOUNDEND (Hunspell) spell-COMPOUNDEND

Probably the same as COMPOUNDLAST

COMPOUNDMIDDLE (Hunspell) spell-COMPOUNDMIDDLE

Words signed with COMPOUNDMIDDLE may be middle elements in
compound words.
Use COMPOUNDRULE instead. spell-COMPOUNDRULE

COMPOUNDRULES (Hunspell) spell-COMPOUNDRULES

Number of COMPOUNDRULE lines following. Ignored, but the
argument must be a number.

COMPOUNDSYLLABLE (Hunspell) spell-COMPOUNDSYLLABLE

Use SYLLABLE and COMPOUNDSYLMAX instead. spell-SYLLABLE
spell-COMPOUNDSYLMAX

KEY (Hunspell) spell-KEY

Define characters that are close together on the keyboard.
Used to give better suggestions. Not supported.

LANG (Hunspell) spell-LANG

This specifies language-specific behavior. This actually
moves part of the language knowledge into the program,
therefore Vim does not support it. Each language property
must be specified separately.

LEMMA_PRESENT (Hunspell) spell-LEMMA_PRESENT

Only needed for morphological analysis.

MAXNGRAMSUGS (Hunspell) spell-MAXNGRAMSUGS

Set number of n-gram suggestions. Not supported.

PSEUDOROOT (Hunspell) spell-PSEUDOROOT

Use NEEDAFFIX instead. spell-NEEDAFFIX

SUGSWITHDOTS (Hunspell) spell-SUGSWITHDOTS

Adds dots to suggestions. Vim doesn't need this.

SYLLABLENUM (Hunspell) spell-SYLLABLENUM

Not supported.

TRY (Myspell, Hunspell, others) spell-TRY

Vim does not use the TRY item, it is ignored. For making
suggestions the actual characters in the words are used, that
is much more efficient.

WORDCHARS (Hunspell) spell-WORDCHARS

Used to recognize words. Vim doesn't need it, because there
is no need to separate words before checking them (using a
trie instead of a hashtable).

5. Spell checker design develop-spell

When spell checking was going to be added to Vim a survey was done over the
available spell checking libraries and programs. Unfortunately, the result
was that none of them provided sufficient capabilities to be used as the spell
checking engine in Vim, for various reasons:

- Missing support for multi-byte encodings. At least UTF-8 must be supported,
  so that more than one language can be used in the same file.
  Doing on-the-fly conversion is not always possible (would require iconv
  support).
- For the programs and libraries: Using them as-is would require installing
  them separately from Vim. That's mostly not impossible, but a drawback.
- Performance: A few tests showed that it's possible to check spelling on the
  fly (while redrawing), just like syntax highlighting. But the mechanisms
  used by other code are much slower. Myspell uses a hashtable, for example.
  The affix compression that most spell checkers use makes it slower too.
- For using an external program like aspell a communication mechanism would
  have to be setup. That's complicated to do in a portable way (Unix-only
  would be relatively simple, but that's not good enough). And performance
  will become a problem (lots of process switching involved).
- Missing support for words with non-word characters, such as "Etten-Leur" and
  "et al.", would require marking the pieces of them OK, lowering the
  reliability.
- Missing support for regions or dialects. Makes it difficult to accept
  all English words and highlight non-Canadian words differently.
- Missing support for rare words. Many words are correct but hardly ever used
  and could be a misspelled often-used word.
- For making suggestions the speed is less important and requiring to install
  another program or library would be acceptable. But the word lists probably
  differ, the suggestions may be wrong words.

Spelling suggestions develop-spell-suggestions

For making suggestions there are two basic mechanisms:
1. Try changing the bad word a little bit and check for a match with a good
   word. Or go through the list of good words, change them a little bit and
   check for a match with the bad word. The changes are deleting a character,
   inserting a character, swapping two characters, etc.
2. Perform soundfolding on both the bad word and the good words and then find
   matches, possibly with a few changes like with the first mechanism.

The first is good for finding typing mistakes. After experimenting with
hashtables and looking at solutions from other spell checkers the conclusion
was that a trie (a kind of tree structure) is ideal for this. Both for
reducing memory use and being able to try sensible changes. For example, when
inserting a character only characters that lead to good words need to be
tried. Other mechanisms (with hashtables) need to try all possible letters at
every position in the word. Also, a hashtable has the requirement that word
boundaries are identified separately, while a trie does not require this.
That makes the mechanism a lot simpler.

Soundfolding is useful when someone knows how the words sounds but doesn't
know how it is spelled. For example, the word "dictionary" might be written
as "daktonerie". The number of changes that the first method would need to
try is very big, it's hard to find the good word that way. After soundfolding
the words become "tktnr" and "tkxnry", these differ by only two letters.

To find words by their soundfolded equivalent (soundalike word) we need a list
of all soundfolded words. A few experiments have been done to find out what
the best method is. Alternatives:
1. Do the sound folding on the fly when looking for suggestions. This means
   walking through the trie of good words, soundfolding each word and
   checking how different it is from the bad word. This is very efficient for
   memory use, but takes a long time. On a fast PC it takes a couple of
   seconds for English, which can be acceptable for interactive use. But for
   some languages it takes more than ten seconds (e.g., German, Catalan),
   which is unacceptable slow. For batch processing (automatic corrections)
   it's too slow for all languages.
2. Use a trie for the soundfolded words, so that searching can be done just
   like how it works without soundfolding. This requires remembering a list
   of good words for each soundfolded word. This makes finding matches very
   fast but requires quite a lot of memory, in the order of 1 to 10 Mbyte.
   For some languages more than the original word list.

3. Like the second alternative, but reduce the amount of memory by using affix
   compression and store only the soundfolded basic word. This is what Aspell
   does. Disadvantage is that affixes need to be stripped from the bad word
   before soundfolding it, which means that mistakes at the start and/or end
   of the word will cause the mechanism to fail. Also, this becomes slow when
   the bad word is quite different from the good word.

The choice made is to use the second mechanism and use a separate file. This way a user with sufficient memory can get very good suggestions while a user who is short of memory or just wants the spell checking and no suggestions doesn't use so much memory.

Word frequency

For sorting suggestions it helps to know which words are common. In theory we could store a word frequency with the word in the dictionary. However, this requires storing a count per word. That degrades word tree compression a lot. And maintaining the word frequency for all languages will be a heavy task. Also, it would be nice to prefer words that are already in the text. This way the words that appear in the specific text are preferred for suggestions.

What has been implemented is to count words that have been seen during displaying. A hashtable is used to quickly find the word count. The count is initialized from words listed in COMMON items in the affix file, so that it also works when starting a new file.

This isn't ideal, because the longer Vim is running the higher the counts become. But in practice it is a noticeable improvement over not using the word count.

Nvim arguments

cli-arguments

Most often, Nvim is started to edit a single file with the command:

```
nvim filename
```

More generally, Nvim is started with:

```
nvim [option | filename] ..
```

Option arguments and file name arguments can be mixed, and any number of them can be given. However, watch out for options that take an argument.

The following items decide how to start editing:

-file --- filename One or more file names. The first one will be the current file and read into the buffer. The cursor will be positioned on the first line of the buffer. To avoid a file name starting with a '-' being interpreted as an option, precede the arglist with "--", e.g.:

```
nvim -- -filename
```

All arguments after "--" are interpreted as file names, no other options or "+command" arguments can follow.

--

- Alias for stdin (standard input). Example:

```
echo text | nvim - file
```

"text" is read into buffer 1, "file" is opened as buffer 2. In most cases (except -s, -es, --embed, --headless) if stdin is not a TTY then it is read as text, so "-" is implied:

```
echo text | nvim file
```

The buffer will be marked as modified, because it contains text that needs to be saved (except for readonly -R mode). If you don't like that, put these lines in your init.vim:

```
" Don't set 'modified' when reading from stdin
au StdinReadPost * set nomodified
```

To read stdin as Normal commands use -s with "-":

```
echo "ifoo" | nvim -s -
```

To read stdin as Ex commands use -es or -e:

```
echo "echo getpid()" | nvim -e - -V1
```

To open a file literally named "-", put it after "--":

```
echo foo | nvim -- -
```

To read stdin as text with --headless use "-".

-t -tag -t {tag} A tag. "tag" is looked up in the tags file, the associated file becomes the current file, and the associated command is executed. Mostly this is used for C programs, in which case "tag" often is a function name. The effect is that the file containing that function becomes the current file and the cursor is positioned on the start of the function (see tags).

-q -qf -q [errorfile] QuickFix mode. The file with the name [errorfile] is read and the first error is displayed. See quickfix. If [errorfile] is not given, the 'errorfile' option is used for the file name. See 'errorfile' for the default value.

(nothing) Without one of the four items above, Vim will start editing a new buffer. It's empty and doesn't have a file name.

startup-options

The option arguments may be given in any order. Single-letter options can be combined after one dash. There can be no option arguments after the "--" argument.

--help -h --help -? -? -h Give usage (help) message and exit.

--version -v --version -v Print version information and exit. Same output as for :version command.

--clean

--clean Mimics a fresh install of Nvim:

- Skips initializations from files and environment variables.
- No 'shada' file is read or written.
- Excludes user directories from 'runtimepath'
- Loads builtin plugins, unlike "-u NONE -i NONE".

--noplugin

--noplugin Skip loading plugins. Resets the 'loadplugins' option. Note that the -u argument may also disable loading plugins: argument load vimrc files load plugins

```
(nothing)               yes                yes
-u NONE                 no                 no
-u NORC                 no                 yes
--noplugin              yes                no
```

--startuptime {fname} --startuptime
During startup write timing messages to the file {fname}. This can be used to find out where time is spent while loading your config, plugins and opening the first file. When {fname} already exists new messages are appended.

-+
+[num] The cursor will be positioned on line "num" for the first file being edited. If "num" is missing, the cursor will be positioned on the last line.

-+/
+/{pat} The cursor will be positioned on the first line containing "pat" in the first file being edited (see pattern for the available search patterns). The search starts at the cursor position, which can be the first line or the cursor position last used from shada. To force a search from the first line use "+1 +/pat".

+{command} -+c -c -c {command} {command} will be executed after the first file has been read (and after autocommands and modelines for that file have been processed). "command" is interpreted as an Ex command. If the "command" contains spaces, it must be enclosed in double quotes (this depends on the shell that is used). Example:

```
vim  "+set si"  main.c
vim  "+find stdio.h"
vim  -c "set ff=dos"  -c wq  mine.mak
```

Note: You can use up to 10 "+" or "-c" arguments in a Vim command. They are executed in the order given. A "-S" argument counts as a "-c" argument as well.

--cmd {command} --cmd
{command} will be executed before processing any vimrc file. Otherwise, it acts like -c {command}. You can use up to 10 of these commands, independently from "-c" commands.

-S
-S [file] Executes Vimscript or Lua (".lua") [file] after the first file has been read. See also :source. If [file] is not given, defaults to "Session.vim". Equivalent to:

```
-c "source {file}"
```

Can be repeated like "-c", subject to the same limit of 10 "-c" arguments. {file} cannot start with a "-".

-L -L -r -r Recovery mode. Without a file name argument, a list of existing swap files is given. With a file name, a swap file is read to recover a crashed editing session. See crash-recovery.

-R
-R Readonly mode. The 'readonly' option will be set for all the files being edited. You can still edit the buffer, but will be prevented from accidentally overwriting a file. If you forgot that you are in View mode and did make some changes, you can overwrite a file by adding an exclamation mark to the Ex command, as in ":w!". The 'readonly' option can be reset with ":set noro" (see the options chapter, options). Subsequent edits will not be done in readonly mode. The 'updatecount' option will be set to 10000, meaning that the swap file will not be updated automatically very often. See -M for disallowing modifications.

-m
-m Modifications not allowed to be written. The 'write' option will be reset, so that writing files is disabled. However, the 'write' option can be set to enable writing again.

-M
-M Modifications not allowed. The 'modifiable' option will be reset, so that changes are not allowed. The 'write' option will be reset, so that writing files is disabled. However, the 'modifiable' and 'write' options can be set to enable changes and writing.

-e -e -E -E Start Nvim in Ex-mode.

If stdin is not a TTY:
- -e reads/executes stdin as Ex commands.
- -E reads stdin as text (into buffer 1).

-es -es -Es -s-ex silent-mode -Es Script mode, aka "silent mode", aka "batch mode". No UI, disables most prompts and messages. Unrelated to -s. See also -S to run script files.

-es reads/executes stdin as Ex commands.

```
printf "put ='foo'\n%%print\n" | nvim -es
```

-Es reads stdin as text (into buffer 1). Use -c or "+" to send commands.

```
printf "foo\n" | nvim -Es +"%print"
```

These commands display on stdout:
- :list
- :number
- :print
- :set

With :verbose or 'verbose', other commands display on stderr:

```
nvim -es +"verbose echo 'foo'"
nvim -V1 -es +"echo 'foo'"
```

Commands that wait for typed input (input(), getchar(), ...) stop Nvim on EOF; they do not consume stdin. Lua can read stdin as data:

```
printf 'lua vim.g.b = io.read()\nfoo\nput =g:b\np' | nvim -es
```

Skips user config unless -u was given. Disables shada unless -i was given. Disables swapfile (like -n).

-l

-l {script} [args] Executes Lua {script} non-interactively (no UI) with optional [args] after processing any preceding Nvim cli-arguments, then exits. Exits 1 on Lua error. See -S to run multiple Lua scripts without args, with a UI.

lua-args

All [args] are treated as {script} arguments and stored in the Lua _G.arg global table, thus "-l" ends processing of Nvim arguments. The {script} name is stored at _G.arg[0].

Sets 'verbose' to 1 (like "-V1"), so Lua print() writes to output, as well as other message-emitting functions like :echo. If {script} prints messages and doesn't cause Nvim to exit, Nvim ensures output ends with a newline.

Arguments before "-l" are processed before executing {script}. This example quits before executing "foo.lua":

```
nvim +q -l foo.lua
```

This loads Lua module "bar" before executing "foo.lua":

```
nvim +"lua require('bar')" -l foo.lua
```

lua-shebang

You can set the "shebang" of the script so that Nvim executes the script when called with "./" from a shell (remember to "chmod u+x"):

```
#!/usr/bin/env -S nvim -l
```

Skips user config unless -u was given. Disables plugins unless 'loadplugins' was set. Disables shada unless -i was given. Disables swapfile (like -n).

-b

-b Binary mode. File I/O will only recognize <NL> to separate lines. The 'expandtab' option will be reset. The 'textwidth' option is set to 0. 'modeline' is reset. The 'binary' option is set. This is done after reading the vimrc but before reading any file in the arglist. See also edit-binary.

-A

-A Arabic mode. Sets the 'arabic' option on.

-H

-H Hebrew mode. Sets the 'rightleft' option on and the 'keymap' option to "hebrew".

-V verbose -V[N] Verbose. Sets the 'verbose' option to [N] (default: 10). Messages will be given for each file that is ":source"d and for reading or writing a ShaDa file. Can be used to find out what is happening upon startup and exit. Example:

```
nvim -V8
```

-V[N]{file} Like -V and sets 'verbosefile' to {file} (must not start with a digit). Messages are not displayed, instead they are written to {file}. Example:

```
nvim -V20vimlog
```

-D

-D Debugging. Go to debugging mode when executing the first command from a script. debug-mode

-n

-n Disables swap-file by setting 'updatecount' to 0 (after executing any vimrc). Recovery after a crash will be impossible. Improves performance when working with a file on a very slow medium (usb drive, network share).

Enable it again by setting 'updatecount' to some value, e.g. ":set updatecount=100".

To reduce accesses to the disk, don't use "-n", but set 'updatetime' and 'updatecount' to very big numbers, and type ":preserve" when you want to save your work. This way you keep the possibility for crash recovery.

-o

-o[N] Open N windows, split horizontally. If [N] is not given, one window is opened for every file given as argument. If there is not enough room, only the first few files get a window. If there are more windows than arguments, the last few windows will be editing an empty file.

-O

-O[N] Open N windows, split vertically. Otherwise, it's like -o. If both the -o and the -O option are given, the last one on the command line determines how the windows will be split.

-p

-p[N] Open N tabpages. If [N] is not given, one tabpage is opened for every file given as argument. The maximum is set with 'tabpagemax' pages (default 50). If there are more tabpages than arguments, the last few tabpages will be editing an empty file. Also see tabpage.

-d

-d Start in diff-mode.

-u E282 -u {vimrc} The file {vimrc} is read for initializations. Most other initializations are skipped; see initialization.

This can be used to start Vim in a special mode, with special mappings and settings. A shell alias can be used to make this easy to use. For example, in a C shell descendant:

```
alias vimc 'nvim -u ~/.config/nvim/c_init.vim \!*'
```

And in a Bash shell:

```
alias vimc='nvim -u ~/.config/nvim/c_init.vim'
```

Also consider using autocommands; see autocommand.

When {vimrc} is "NONE" (all uppercase), all initializations from files and environment variables are skipped. Plugins and syntax highlighting are also skipped.

When {vimrc} is "NORC" (all uppercase), this has the same effect as "NONE", but plugins and syntax highlighting are not skipped.

-i

-i {shada} The file {shada} is used instead of the default ShaDa file. If the name "NONE" is used (all uppercase), no ShaDa file is read or written, even if 'shada' is set or when ":rsh" or ":wsh" are used. See also shada-file.

-s

-s {scriptin} Read script file {scriptin}, interpreting characters as Normal-mode input. The same can be done with ":source!":

```
:source! {scriptin}
```

Reads from stdin if {scriptin} is "-":

```
echo "ifoo" | nvim -s -
```

If the end of the file is reached before Nvim exits, further characters are read from the keyboard.

Does not work with -es. See also complex-repeat.

-w_nr

-w {number} -w{number} Set the 'window' option to {number}.

-w

-w {scriptout} All keys that you type are recorded in the file "scriptout", until you exit Vim. Useful to create a script file to be used with "vim -s" or ":source!". Appends to the "scriptout" file if it already exists. {scriptout} cannot start with a digit. See also vim.on_key(). See also complex-repeat.

-W

-W {scriptout} Like -w, but do not append, overwrite an existing file.

--api-info

--api-info Print msgpack-encoded api-metadata and exit.

--embed

--embed Use stdin/stdout as a msgpack-RPC channel, so applications can embed and control Nvim via the RPC API. Nvim exits if the channel is closed, unless the channel was detached (:detach, :detach!, nvim_chan_set()) or another detach=false channel is still open.

Waits for a client to call nvim_ui_attach() before sourcing startup files and reading buffers, so UIs can display early messages/dialogs and make preliminary requests (e.g. nvim_chan_set to set channel config, or nvim_get_api_info for feature-detection). During this pre-startup phase the user config is of course not available (similar to --cmd).

Non-UI embedders must pass --headless, then startup will continue without waiting for nvim_ui_attach:

```
nvim --embed --headless
```

which is equivalent to:

```
nvim --headless --cmd "call stdioopen({'rpc': v:true})"
```

UI embedders that want the UI protocol on a socket (instead of stdio) must pass --listen as well as --embed:

```
nvim --embed --listen addr
```

Then stdin is an RPC channel only if it is a pipe/socket. Otherwise (tty, /dev/null) Nvim waits for a UI to attach at addr, e.g. to start a server for a GUI on a remote host:

```
ssh host 'nvim --embed --listen 0.0.0.0:8888 &'
```

The UI can call nvim_chan_set() with detach=false to enable exit-on-channel-close behavior.

To "daemonize" such a server, use jobstart() with "detach":

```
nvim -es +"lua vim.fn.jobstart({'nvim', '--embed', '--listen', 'addr'}, {detach=true, stdin='null'})" +q
```

See also: ui-startup channel-stdio

--headless

--headless Start without UI, and do not wait for nvim_ui_attach. The builtin TUI is not used, so stdio works as an arbitrary communication channel. channel-stdio

Also useful for scripting (tests) to see messages that would not be printed by -es.

To detect if a UI is available, check if nvim_list_uis() is empty during or after VimEnter.

To read stdin as text, "-" must be given explicitly: --headless cannot assume that stdin is just text.

```
echo foo | nvim --headless +"%print" +"q!" -
```

See also --embed. See also -es, which also disables most messages.

--listen {addr} --listen
Start RPC server on pipe or TCP address {addr}. Sets the primary listen address v:servername to {addr}. serverstart()

To start the server on-demand with systemd, use a systemd socket unit and associated service unit running:

```
systemd-socket-proxyd --exit-idle-time
```

Initialization initialization startup

At startup, Nvim checks environment variables and files and sets values accordingly, proceeding as follows:

1. Set v:starttime and v:startreason.

2. Set the 'shell' option. SHELL COMSPEC The environment variable SHELL, if it exists, is used to set the 'shell' option. On Win32, the COMSPEC variable is used if SHELL is not set.
3. Process the arguments. The options and file names from the command that start Vim are inspected. The -V argument can be used to display or log what happens next, useful for debugging the initializations. The --cmd arguments are executed. Buffers are created for all files (but not loaded yet).
4. Start a server (unless --listen was given) and set v:servername.
5. Wait for UI to connect. Nvim started with --embed waits for the UI to connect before proceeding to load user configuration.
6. Setup default-mappings and default-autocmds. Create popup-menu.
7. Enable filetype and indent plugins. Skipped if the "-u NONE" command line argument was given. This does the same as:

```
:runtime! ftplugin.vim indent.vim
```

8. Load user config (execute Ex commands from files, environment, ...). $VIMINIT environment variable is read as one Ex command line (separate multiple commands with '|' or <NL>). config init.vim init.lua vimrc exrc
   - A file containing initialization commands is generically called a "vimrc" or config file. It can be Vimscript ("init.vim") or Lua ("init.lua"), but not both. E5422 See vimrc-intro base-directories. The config file is located at:
     - Unix ~/.config/nvim/init.vim (or init.lua)
     - Windows ~/AppData/Local/nvim/init.vim (or init.lua)
     - $XDG_CONFIG_HOME $XDG_CONFIG_HOME/nvim/init.vim (or init.lua)
   - If Nvim was started with "-u `{file}`" then `{file}` is used as the config and all initializations until 8. are skipped. $MYVIMRC is not set. "nvim -u NORC" can be used to skip these initializations without reading a file. "nvim -u NONE" also skips plugins and syntax highlighting. -u
   - If Nvim was started with -es or -Es or -l all initializations until "8" are skipped.
   - a. system-vimrc sysinit.vim The system vimrc file is read for initializations. If nvim/sysinit.vim file exists in one of $XDG_CONFIG_DIRS, it will be used. Otherwise the system vimrc file is used. The path of this file is given by the :version command. Usually it's "$VIM/sysinit.vim".
   - b. VIMINIT EXINIT $MYVIMRC Locations searched for initializations, in order of preference. $MYVIMRC is set to the first valid location unless it was already set or when using $VIMINIT:
     - $VIMINIT environment variable (Ex command line).
     - User config: $XDG_CONFIG_HOME/nvim/init.vim (or init.lua).
     - Other config: `{dir}`/nvim/init.vim (or init.lua) where `{dir}` is any directory in $XDG_CONFIG_DIRS.
     - $EXINIT environment variable (Ex command line).
   - c. If the 'exrc' option is on, the current-directory is searched until one of these files is found, in order of precedence:
     - ".nvim.lua"
     - ".nvimrc"
     - ".exrc"
9. Enable filetype detection. Skipped if ":filetype off" was called or if the "-u NONE" command line argument was given. This does the same as:

```
:runtime! filetype.lua
```

10. Enable syntax highlighting. Skipped if ":syntax off" was called or if the "-u NONE" command line argument was given. This does the same as:

```
:runtime! syntax/syntax.vim
```

11. Set the v:vim_did_init variable to 1.
12. Load the plugin scripts. load-plugins
    This does the same as: `:runtime! plugin/**/*.{vim,lua}`
    - The result is that all directories in 'runtimepath' will be searched for the "plugin" sub-directory and all files ending in ".vim" or ".lua" will be sourced (in alphabetical order per directory), also in subdirectories. First "*.vim" are sourced, then "*.lua" files, per directory.
    - However, directories in 'runtimepath' ending in "after" are skipped here and only loaded after packages, see below.
    - Loading plugins won't be done when:
      - The 'loadplugins' option was reset in a vimrc file.
      - The --noplugin command line argument is used.
      - The --clean command line argument is used.
      - The "-u NONE" command line argument is used -u.
    - Note that using `-c 'set noloadplugins'` doesn't work, because the commands from the command line have not been executed yet. You can use `--cmd 'set noloadplugins'` or `--cmd 'set loadplugins'` --cmd.
    - Packages are loaded. These are plugins, as above, but found in the "start" directory of each entry in 'packpath'. Every plugin directory found is added in 'runtimepath' and then the plugins are sourced. See packages.
    - The plugins scripts are loaded, as above, but now only the directories ending in "after" are used. Note that 'runtimepath' will have changed if packages have been found, but that should not add a directory ending in "after".
13. Set 'shellpipe' and 'shellredir', according to the value of the 'shell' option, unless they were already set. Nvim will figure out the values of 'shellpipe' and 'shellredir' for you, unless you have set them yourself.
14. Set 'updatecount' to zero, if -n was given.
15. Set binary options if -b was given.
16. Read the shada-file.
17. Read the quickfix file if -q was given, or exit on failure.
18. Open all windows.
    - When -o was given, windows will be opened (but not displayed yet).
    - When -p was given, tabpages will be created (but not displayed yet).
    - When switching screens, it happens now. Redrawing starts.
    - If -q was given, the first error is jumped to.
    - Buffers for all windows will be loaded, without triggering BufAdd.
19. Execute startup commands.
    - If -t was given, the tag is jumped to.
    - Commands given with -c and +cmd are executed.

- The vim_starting flag is reset, `has("vim_starting")` will now return zero.
- The v:vim_did_enter variable is set to 1.
- The VimEnter autocommands are executed.

Saving the current state of Vim to a file

Whenever you have changed values of options or when you have created a mapping, then you may want to save them in a vimrc file for later use. See save-settings about saving the current state of settings to a file.

Avoiding trojan horses

trojan-horse

While reading the "vimrc" or the "exrc" file in the current directory, some commands can be disabled for security reasons by setting the 'secure' option. This is always done when executing the command from a tags file. Otherwise, it would be possible that you accidentally use a vimrc or tags file that somebody else created and contains nasty commands. The disabled commands are the ones that start a shell, the ones that write to a file, and ":autocmd". The ":map" commands are echoed, so you can see which keys are being mapped.

If you want Vim to execute all commands in a local vimrc file, you can reset the 'secure' option in the EXINIT or VIMINIT environment variable or in the global exrc or vimrc file. This is not possible in vimrc or exrc in the current directory, for obvious reasons.

On Unix systems, this only happens if you are not the owner of the vimrc file. **Warning:** If you unpack an archive that contains a vimrc or exrc file, it will be owned by you. You won't have the security protection. Check the vimrc file before you start Vim in that directory, or reset the 'exrc' option. Some Unix systems allow a user to do "chown" on a file. This makes it possible for another user to create a nasty vimrc and make you the owner. Be careful!

When using tag search commands, executing the search command (the last part of the line in the tags file) is always done in secure mode. This works just like executing a command from a vimrc in the current directory.

If Vim startup is slow

slow-start

If Vim takes a long time to start up, use the --startuptime argument to find out what happens.

If you have 'shada' enabled, the loading of the ShaDa file may take a while. You can find out if this is the problem by disabling ShaDa for a moment (use the Vim argument "-i NONE", -i). Try reducing the number of lines stored in a register with ":set shada='20,<50,s10". shada-file.

Troubleshooting broken configurations

bisect

The extreme flexibility of editors like Vim and Emacs means that any plugin or setting can affect the entire editor in ways that are not initially obvious.

To find the cause of a problem in your config, you must "bisect" it:

1. Remove or disable half of your config.
2. Restart Nvim (:restart).
3. If the problem still occurs, goto 1.
4. If the problem is gone, restore half of the removed lines.
5. Continue narrowing your config in this way, until you find the setting or plugin causing the issue.

Intro message

:intro

When Vim starts without a file name, an introductory message is displayed. It is removed as soon as the display is redrawn. To see the message again, use the ":intro" command. To avoid the intro message on startup, add the "I" flag to 'shortmess'.

$VIM and $VIMRUNTIME

$VIM

The environment variable "$VIM" is used to locate various user files for Nvim, such as the user config. This depends on the system, see startup.

Nvim will try to get the value for $VIM in this order:

1. Environment variable $VIM, if it is set.
2. Path derived from the 'helpfile' option, unless it contains some environment variable too (default is "$VIMRUNTIME/doc/help.txt"). File name ("help.txt", etc.) is removed. Trailing directory names are removed, in this order: "doc", "runtime".
3. Path derived from the location of the `nvim` executable.
4. Compile-time defined installation directory (see output of ":version").

After doing this once, Nvim sets the $VIM environment variable.

$VIMRUNTIME

The environment variable "$VIMRUNTIME" is used to locate various support files, such as the documentation and syntax-highlighting files. For example, the main help file is normally "$VIMRUNTIME/doc/help.txt".

Nvim will try to get the value for $VIMRUNTIME in this order:

1. Environment variable $VIMRUNTIME, if it is set.
2. Directory path "$VIM/runtime", if it exists.
3. Value of $VIM environment variable. This is for backwards compatibility with older Vim versions.
4. If "../share/nvim/runtime" exists relative to v:progpath, it is used.
5. Path derived from the 'helpfile' option (if it doesn't contain '$') with "doc/help.txt" removed from the end.

After doing this once, Nvim sets the $VIMRUNTIME environment variable.

In case you need the value of $VIMRUNTIME in a shell (e.g., for a script that greps in the help files) you might be able to use this:

```
VIMRUNTIME="$(nvim --clean --headless --cmd 'echo $VIMRUNTIME|q')"
```

Suspending **suspend**

CTRL-Z v_CTRL-Z CTRL-Z Suspend Nvim, like ":stop". Works in Normal and in Visual mode. In Insert and Command-line mode, the CTRL-Z is inserted as a normal character. In Visual mode Nvim goes back to Normal mode before suspending.

:sus[pend][!] or :sus :suspend :st :stop :st[op][!] Suspend Nvim using OS "job control"; it will continue if you make it the foreground job again. Triggers VimSuspend before suspending and VimResume when resumed.

If "!" is not given and 'autowrite' is set, every buffer with changes and a file name is written out. If "!" is given or 'autowrite' is not set, changed buffers are not written, don't forget to bring Nvim back to the foreground later!

In the GUI, suspending is implementation-defined.

Exiting **exiting**

There are several ways to exit Vim:
- Close the last window with `:quit`. Only when there are no changes.
- Close the last window with `:quit!`. Also when there are changes.
- Close all windows with `:qall`. Only when there are no changes.
- Close all windows with `:qall!`. Also when there are changes.
- Use `:cquit`. Also when there are changes.

When using `:cquit` or when there was an error message Vim exits with exit code 1. Errors can be avoided by using `:silent!` or with `:catch`.

Saving settings **save-settings**

Mostly you will edit your vimrc files manually. This gives you the greatest flexibility. There are a few commands to generate a vimrc file automatically. You can use these files as they are, or copy/paste lines to include in another vimrc file.

:mk :mkexrc :mk[exrc] [file] Write current key mappings and changed options to [file] (default ".exrc" in the current directory), unless it already exists.

:mk[exrc]! [file] Always write current key mappings and changed options to [file] (default ".exrc" in the current directory).

:mkv :mkvi :mkvimrc :mkv[imrc][!] [file] Like ":mkexrc", but the default is ".nvimrc" in the current directory. The ":version" command is also written to the file.

These commands will write ":map" and ":set" commands to a file, in such a way that when these commands are executed, the current key mappings and options will be set to the same values. The options 'columns', 'endofline', 'fileformat', 'lines', 'modified', and 'scroll' are not included, because these are terminal or file dependent. Note that the options 'binary', 'paste' and 'readonly' are included, this might not always be what you want.

When special keys are used in mappings, the 'cpoptions' option will be temporarily set to its Vim default, to avoid the mappings to be misinterpreted. This makes the file incompatible with Vi, but makes sure it can be used with different terminals.

Only global mappings are stored, not mappings local to a buffer.

A common method is to use a default config file, make some modifications with ":map" and ":set" commands and write the modified file. First read the default vimrc in with a command like ":source ~piet/.vimrc.Cprogs", change the settings and then save them in the current directory with ":mkvimrc!". If you want to make this file your default config, move it to $XDG_CONFIG_HOME/nvim. You could also use autocommands autocommand and/or modelines modeline.

vimrc-option-example

If you only want to add a single option setting to your vimrc, you can use these steps:
1. Edit your vimrc file with Vim.
2. Play with the option until it's right. E.g., try out different values for 'guifont'.
3. Append a line to set the value of the option, using the expression register '=' to enter the value. E.g., for the 'guifont' option:

```
o:set guifont=<C-R>=&guifont<CR><Esc>
```

[<C-R> is a CTRL-R, <CR> is a return, <Esc> is the escape key] You need to escape special characters, esp. spaces.

Views and Sessions **views-sessions**

This is introduced in sections 21.4 and 21.5 of the user manual.

View view-file A View is a collection of settings that apply to one window. You can save a View and when you restore it later, the text is displayed in the same way. The options and mappings in this window will also be restored, so that you can continue editing like when the View was saved.

Session session-file A Session keeps the Views for all windows, plus the global settings. You can save a Session and when you restore it later the window layout looks the same. You can use a Session to quickly switch between different projects, automatically loading the files you were last working on in that project.

Views and Sessions are a nice addition to ShaDa files, which are used to remember information for all Views and Sessions together shada-file.

You can quickly start editing with a previously saved View or Session with the -S argument:

```
vim -S Session.vim
```

:mks :mksession :mks[ession][!] [file] Write a Vim script that restores the current editing session. When [!] is included, an existing file is overwritten. When [file] is omitted, "Session.vim" is used.

The output of ":mksession" is like ":mkvimrc", but additional commands are added to the file. Which ones depends on the 'sessionoptions' option. The resulting file, when executed with a ":source" command:
1. Restores global mappings and options, if 'sessionoptions' contains "options". Lua functions, and script-local mappings, will not be written.

2. Restores global variables that start with an uppercase letter and contain at least one lowercase letter, if 'sessionoptions' contains "globals".
3. Closes all windows in the current tabpage, except the current one; closes all tabpages except the current one (this results in currently loaded buffers to be unloaded, some may become hidden if 'hidden' is set or otherwise specified); wipes out the current buffer, if it is empty and unnamed.
4. Restores the current directory, if 'sessionoptions' contains "curdir", or sets the current directory to where the Session file is, if 'sessionoptions' contains "sesdir".
5. Restores GUI Vim window position, if 'sessionoptions' contains "winpos".
6. Restores screen size, if 'sessionoptions' contains "resize".
7. Reloads the buffer list, with the last cursor positions. If 'sessionoptions' contains "buffers" then all buffers are restored, including hidden and unloaded buffers. Otherwise, only buffers in windows are restored.
8. Restores all windows with the same layout. If 'sessionoptions' contains "help", help windows are restored. If 'sessionoptions' contains "blank", windows editing a buffer without a name will be restored. If 'sessionoptions' contains "winsize" and no (help/blank) windows were left out, the window sizes are restored (relative to the screen size). Otherwise, the windows are just given sensible sizes.
9. Restores the Views for all the windows, as with :mkview. But 'sessionoptions' is used instead of 'viewoptions'.
10. If a file exists with the same name as the Session file, but ending in "x.vim" (for eXtra), executes that as well. You can use `*x.vim` files to specify additional settings and actions associated with a given Session, such as creating menu items in the GUI version.

After restoring the Session, the full filename of your current Session is available in the internal variable v:this_session. An example mapping:

```
:nmap <F2> :wa<Bar>exe "mksession! " .. v:this_session<CR>:so ~/sessions/
```

This saves the current Session, and starts off the command to load another.

A session includes all tabpages, unless "tabpages" was removed from 'sessionoptions'. tabpage

The SessionLoadPre autocmd event is triggered before a session file is loaded/sourced and SessionLoadPost autocmd event is triggered after. While the session file is loading, `g:SessionLoad` is set to 1. Plugins can use this to postpone some work until the SessionLoad-variable SessionLoadPost event is triggered.

:mkvie :mkview :mkvie[w][!] [file] Write a Vim script that restores the contents of the current window. When [!] is included, an existing file is overwritten. When [file] is omitted or is a number from 1 to 9, a name is generated and 'viewdir' prepended. When the last path part of 'viewdir' does not exist, this directory is created. E.g., when 'viewdir' is "$VIM/vimfiles/view" then "view" is created in "$VIM/vimfiles". An existing file is always overwritten then. Use :loadview to load this view again. When [file] is the name of a file ('viewdir' is not used), a command to edit the file is added to the generated file.

The output of ":mkview" contains these items:
1. The argument list used in the window. When the global argument list is used, it is reset to the global list. The index in the argument list is also restored.
2. The file being edited in the window. If there is no file, the window is made empty.
3. Restore mappings, abbreviations and options local to the window, if 'viewoptions' contains "options" or "localoptions". Only option values that are local to the current buffer and the current window are restored. When storing the view as part of a session and "options" is in 'sessionoptions', global values for local options will be stored too. Lua functions are not restored.
4. Restore folds when using manual folding and 'viewoptions' contains "folds". Restore manually opened and closed folds.
5. The scroll position and the cursor position in the file. Doesn't work very well when there are closed folds.
6. The local current directory, if it is different from the global current directory and 'viewoptions' contains "curdir".

Note that Views and Sessions are not perfect:
- They don't restore everything. For example, defined functions, autocommands and ":syntax on" are not included. Things like register contents and command line history are in ShaDa, not in Sessions or Views.
- Global option values are only set when they differ from the default value. When the current value is not the default value, loading a Session will not set it back to the default value. Local options will be set back to the default value though.
- Existing mappings will be overwritten without warning. An existing mapping may cause an error for ambiguity.
- When storing manual folds and when storing manually opened/closed folds, changes in the file between saving and loading the view will mess it up.
- The Vim script is not very efficient. But still faster than typing the commands yourself!

:lo :loadview :lo[adview] [nr] Load the view for the current file. When [nr] is omitted, the view stored with ":mkview" is loaded. When [nr] is specified, the view stored with ":mkview [nr]" is loaded.

The combination of ":mkview" and ":loadview" can be used to store up to ten different views of a file. These are remembered in the directory specified with the 'viewdir' option. The views are stored using the file name. If a file is renamed or accessed through a (symbolic) link, the view will not be found.

You might want to clean up your 'viewdir' directory now and then.

To automatically save and restore views for `*.c` files:

```
au BufWinLeave *.c mkview
au BufWinEnter *.c silent! loadview
```

Shada ("shared data") file

shada shada-file

If you exit Vim and later start it again, you would normally lose a lot of information. The ShaDa file can be used to remember that information, which enables you to continue where you left off. Its name is the abbreviation of SHAred DAta because it is used for sharing data between Nvim sessions.

This is introduced in section 21.3 of the user manual.

The ShaDa file is used to store:
- The command line history.
- The search string history.
- The input-line history.
- Contents of non-empty registers.

- Marks for several files.
- File marks, pointing to locations in files.
- Last search/substitute pattern (for 'n' and '&').
- The buffer list.
- Global variables.

You could also use a Session file. The difference is that the ShaDa file does not depend on what you are working on. There normally is only one ShaDa file. Session files are used to save the state of a specific editing Session. You could have several Session files, one for each project you are working on. ShaDa and Session files together can be used to effectively enter Vim and directly start working in your desired setup. session-file

shada-read

When Vim is started and the 'shada' option is non-empty, the contents of the ShaDa file are read and the info can be used in the appropriate places. The v:oldfiles variable is filled. The marks are not read in at startup (but file marks are). See initialization for how to set the 'shada' option upon startup.

shada-write

When Vim exits and 'shada' is non-empty, the info is stored in the ShaDa file (it's actually merged with the existing one, if one exists shada-merging). The 'shada' option is a string containing information about what info should be stored, and contains limits on how much should be stored (see 'shada').

Notes for Unix:
- The file protection for the ShaDa file will be set to prevent other users from being able to read it, because it may contain any text or commands that you have worked with.
- If you want to share the ShaDa file with other users (e.g. when you "su" to another user), you can make the file writable for the group or everybody. Vim will preserve this when writing new ShaDa files. Be careful, don't allow just anybody to read and write your ShaDa file!
- Vim will not overwrite a ShaDa file that is not writable by the current "real" user. This helps for when you did "su" to become root, but your $HOME is still set to a normal user's home directory. Otherwise, Vim would create a ShaDa file owned by root that nobody else can read.
- The ShaDa file cannot be a symbolic link. This is to avoid security issues.

Marks are stored for each file separately. When a file is read and 'shada' is non-empty, the marks for that file are read from the ShaDa file. **NOTE:** The marks are only written when exiting Vim, which is fine because marks are remembered for all the files you have opened in the current editing session, unless ":bdel" is used. If you want to save the marks for a file that you are about to abandon with ":bdel", use ":wsh". The '[' and ']' marks are not stored, but the '"' mark is. The '"' mark is very useful for jumping to the cursor position when the file was last exited. No marks are saved for files that start with any string given with the "r" flag in 'shada'. This can be used to avoid saving marks for files on removable media (for MS-Windows you would use "ra:,rb:"). The v:oldfiles variable is filled with the file names that the ShaDa file has marks for.

shada-file-marks

Uppercase marks ('A to 'Z) are stored when writing the ShaDa file. The numbered marks ('0 to '9) are a bit special. When the ShaDa file is written (when exiting or with the :wshada command), '0 is set to the current cursor position and file. The old '0 is moved to '1, '1 to '2, etc. This resembles what happens with the "1 to "9 delete registers. If the current cursor position is already present in '0 to '9, it is moved to '0, to avoid having the same position twice. The result is that with "'0", you can jump back to the file and line where you exited Vim. To do that right away, try using this command:

```
vim -c "normal '0"
```

In a C shell descendant, you could make an alias for it:

```
alias lvim vim -c '"'normal "'"0'"'
```

For a Bash-like shell:

```
alias lvim='vim -c "normal '\''0"'
```

Use the "r" flag in 'shada' to specify for which files no marks should be remembered.

MERGING

shada-merging

When writing ShaDa files with :wshada without bang or at regular exit information in the existing ShaDa file is merged with information from current Nvim instance. For this purpose ShaDa files store timestamps associated with ShaDa entries. Specifically the following is being done:

1. History lines are merged, ordered by timestamp. Maximum amount of items in ShaDa file is defined by 'shada' option (shada-/, shada-:, shada-@, etc: one suboption for each character that represents history name (:history)).
2. Local marks and changes for files that were not opened by Nvim are copied to new ShaDa file. Marks for files that were opened by Nvim are merged, changes to files opened by Nvim are ignored. shada-'
3. Jump list is merged: jumps are ordered by timestamp, identical jumps (identical position AND timestamp) are squashed.
4. Search patterns and substitute strings are not merged: search pattern or substitute string which has greatest timestamp will be the only one copied to ShaDa file.
5. For each register entity with greatest timestamp is the only saved. shada-<
6. All saved variables are saved from current Nvim instance. Additionally existing variable values are copied, meaning that the only way to remove variable from a ShaDa file is either removing it by hand or disabling writing variables completely. shada-!
7. For each global mark entity with greatest timestamp is the only saved.
8. Buffer list and header are the only entries which are not merged in any fashion: the only header and buffer list present are the ones from the Nvim instance which was last writing the file. shada-%

COMPATIBILITY

shada-compatibility

ShaDa files are forward and backward compatible. This means that

1. Entries which have unknown type (i.e. that hold unidentified data) are ignored when reading and blindly copied when writing.
2. Register entries with unknown register name are ignored when reading and blindly copied when writing. Limitation: only registers that use name with code in interval [1, 255] are supported. registers

3. Register entries with unknown register type are ignored when reading and merged as usual when writing. getregtype()
4. Local and global mark entries with unknown mark names are ignored when reading. When writing global mark entries are blindly copied and local mark entries are also blindly copied, but only if file they are attached to fits in the shada-' limit. Unknown local mark entry's timestamp is also taken into account when calculating which files exactly should fit into this limit. Limitation: only marks that use name with code in interval [1, 255] are supported. mark-motions
5. History entries with unknown history type are ignored when reading and blindly copied when writing. Limitation: there can be only up to 256 history types. history
6. Unknown keys found in register, local mark, global mark, change, jump and search pattern entries are saved internally and dumped when writing. Entries created during Nvim session never have such additions.
7. Additional elements found in replacement string and history entries are saved internally and dumped. Entries created during Nvim session never have such additions.
8. Additional elements found in variable entries are simply ignored when reading. When writing new variables they will be preserved during merging, but that's all. Variable values dumped from current Nvim session never have additional elements, even if variables themselves were obtained by reading ShaDa files.

"Blindly" here means that there will be no attempts to somehow merge them, even if other entries (with known name/type/etc) are merged. shada-merging

SHADA FILE NAME **shada-file-name**

- Default name of the shada file is: Unix: "$XDG_STATE_HOME/nvim/shada/main.shada" Windows: "$XDG_STATE_HOME/nvim-data/shada/main.shada" See also base-directories.
- To choose a different file name you can use:
  - The "n" flag in the 'shada' option.
  - The -i startup argument. "NONE" means no shada file is ever read or written. Also not for the commands below!
  - The 'shadafile' option. The value from the "-i" argument (if any) is stored in the 'shadafile' option.
- For the commands below, another file name can be given, overriding the default and the name given with 'shada' or "-i" (unless it's NONE).

MANUALLY READING AND WRITING **shada-read-write**

Two commands can be used to read and write the ShaDa file manually. This can be used to exchange registers between two running Vim programs: First type ":wsh" in one and then ":rsh" in the other. Note that if the register already contained something, then ":rsh!" would be required. Also note, however, that this means everything will be overwritten with information from the first Vim, including the command line history, etc.

The ShaDa file itself can be edited by hand too, although we suggest you start with an existing one to get the format right. You need to understand MessagePack (or, more likely, find software that is able to use it) format to do this. This can be useful in order to create a second file, say "~/.my.shada", which could contain certain settings that you always want when you first start Nvim. For example, you can preload registers with particular data, or put certain commands in the command line history. A line in your config file like

```
:rshada! ~/.my.shada
```

can be used to load this information. You could even have different ShaDa files for different types of files (e.g., C code) and load them based on the file name, using the ":autocmd" command (see :autocmd). More information on ShaDa file format is contained in shada-format section.

E136 E929 shada-error-handling Some errors make Nvim leave temporary file named `{basename}.tmp.X` (X is any free letter from a to z) while normally it will create this file, write to it and then rename `{basename}.tmp.X` to `{basename}`. Such errors include:

- Errors which make Nvim think that the file being read is not a ShaDa file at all: non-ShaDa files are not overwritten for safety reasons to avoid accidentally destroying an unrelated file. This could happen e.g. when typing "nvim -i file" in place of "nvim -R file" (yes, somebody did that at least with Vim). Such errors are listed at shada-critical-contents-errors.
- If writing to the temporary file failed: e.g. because of the insufficient space left.
- If renaming file failed: e.g. because of insufficient permissions.
- If target ShaDa file has different from the Nvim instance's owners (user and group) and changing them failed. Unix-specific, applies only when Nvim was launched from root.

Do not forget to remove the temporary file or replace the target file with temporary one after getting one of the above errors or all attempts to create a ShaDa file may fail with E929. If you got one of them when using :wshada (and not when exiting Nvim: i.e. when you have Nvim session running) you have additional options:

- First thing which you should consider if you got any error, except failure to write to the temporary file: remove existing file and replace it with the temporary file. Do it even if you have running Nvim instance.
- Fix the permissions and/or file ownership, free some space and attempt to write again. Do not remove the existing file.
- Use :wshada with bang. Does not help in case of permission error. If target file was actually the ShaDa file some information may be lost in this case. To make the matters slightly better use :rshada prior to writing, but this still will loose buffer-local marks and change list entries for any file which is not opened in the current Nvim instance.
- Remove the target file from shell and use :wshada. Consequences are not different from using :wshada with bang, but "rm -f" works in some cases when you don't have write permissions.

:rsh :rshada E886 :rsh[ada][!] [file] Read from ShaDa file [file] (default: see above). If [!] is given, then any information that is already set (registers, marks, v:oldfiles, etc.) will be overwritten.

:wsh :wshada E137 E138 :wsh[ada][!] [file] Write to ShaDa file [file] (default: see above). The information in the file is first read in to make a merge between old and new info. When [!] is used, the old information is not read first, only the internal info is written (also disables safety checks described in shada-error-handling). If 'shada' is empty, marks for up to 100 files will be written. When you get error "E929: All .tmp.X files exist, cannot write ShaDa file!", check that no old temp files were left behind (e.g. ~/.local/state/nvim/shada/main.shada.tmp*).

Note: Executing :wshada will reset all 'quote marks.

:o :ol :oldfiles :o[ldfiles] List the files that have marks stored in the ShaDa file. This list is read on startup and only changes afterwards with `:rshada!`. Also see v:oldfiles. The number can be used with c_#<. The output can be filtered with :filter, e.g.:

```
filter /\.vim/ oldfiles
```

The filtering happens on the file name.

:bro[wse] o[ldfiles][!] List file names as with :oldfiles, and then prompt for a number. When the number is valid that file from the list is edited.

Use [!] to abandon a modified buffer. abandon

SHADA FILE FORMAT shada-format

ShaDa files are concats of MessagePack entries. Each entry is a concat of exactly four MessagePack objects:

1. First goes type of the entry. Object type must be an unsigned integer. Object type must not be equal to zero.
2. Second goes entry timestamp. It must also be an unsigned integer.
3. Third goes the length of the fourth entry. Unsigned integer as well, used for fast skipping without parsing.
4. Fourth is actual entry data. All currently used ShaDa entries use containers to hold data: either map or array. All string values in those containers are either binary (applies to filenames) or UTF-8, yet parser needs to expect that invalid bytes may be present in a UTF-8 string.

Exact format depends on the entry type:

Entry type (name) Entry data

- 1 (Header) Map containing data that describes the generator instance that wrote this ShaDa file. It is ignored when reading ShaDa files. Contains the following data: Key Data

```
generator  Binary, software used to generate ShaDa
           file. Is equal to "nvim" when ShaDa file was
           written by Nvim.
version    Binary, generator version.
encoding   Binary, effective 'encoding' value.
max_kbyte  Integer, effective |shada-s| limit value.
pid        Integer, instance process ID.
*          It is allowed to have any number of
           additional keys with any data.
```

- 2 (SearchPattern) Map containing data describing last used search or substitute pattern. Normally ShaDa file contains two such entries: one with "ss" key set to true (describes substitute pattern, see :substitute), and one set to false (describes search pattern, see search-commands). "su" key should be true on one of the entries. If key value is equal to default then it is normally not present. Keys: Key Type Default Description

```
sm   Boolean  true     Effective 'magic' value.
sc   Boolean  false    Effective 'smartcase' value.
sl   Boolean  true     True if search pattern comes
                       with a line offset.  See
                       |search-offset|.
se   Boolean  false    True if |search-offset|
                       requested to place cursor at
                       (relative to) the end of the
                       pattern.
so   Integer  0        Offset value. |search-offset|
su   Boolean  false    True if current entry was the
                       last used search pattern.
ss   Boolean  false    True if current entry describes
                       |:substitute| pattern.
sh   Boolean  false    True if |v:hlsearch| is on.
                       With |shada-h| or 'nohlsearch'
                       this key is always false.
sp   Binary   N/A      Actual pattern.  Required.
sb   Boolean  false    True if search direction is
                       backward.
*    any      none     Other keys are allowed for
                       compatibility reasons, see
                       |shada-compatibility|.
```

- 3 (SubString) Array containing last :substitute replacement string. Contains single entry: binary, replacement string used. More entries are allowed for compatibility reasons, see shada-compatibility.
- 4 (HistoryEntry) Array containing one entry from history. Should have two or three entries. First one is history type (unsigned integer), second is history line (binary), third is the separator character (unsigned integer, must be in interval [0, 255]). Third item is only valid for search history. Possible history types are listed in hist-names, here are the corresponding numbers: 0 - cmd, 1 - search, 2 - expr, 3 - input, 4 - debug.
- 5 (Register) Map describing one register (registers). If key value is equal to default then it is normally not present. Keys: Key Type Def Description

```
rt   UInteger         0     Register type:
                           No  Description ~
                           0   |charwise-register|
                           1   |linewise-register|
                           2   |blockwise-register|
rw   UInteger         0     Register width. Only valid
                           for |blockwise-register|s.
rc   Array of binary  N/A   Register contents.  Each
                           entry in the array
                           represents its own line.
                           NUL characters inside the
                           line should be represented
                           as NL according to
                           |NL-used-for-Nul|.
ru   Boolean          false Unnamed register. Whether
                           the unnamed register had
                           pointed to this register.
n    UInteger         N/A   Register name: character
                           code in range [1, 255].
                           Example: |quote0| register
                           has name 48 (ASCII code for
                           zero character).
*    any              none  Other keys are allowed
                           for compatibility reasons,
                           see |shada-compatibility|.
```

- 6 (Variable) Array containing two items: variable name (binary) and variable value (any object). Values are converted using the same code msgpackparse() uses when reading, msgpackdump() when writing, so there may appear msgpack-special-dicts. If there are more then two entries then the rest are ignored (shada-compatibility).
- 7 (GlobalMark)
- 8 (Jump)
- 10 (LocalMark)
- 11 (Change) Map containing some position description: Entry Position

```
GlobalMark Global mark position. |'A|
LocalMark  Local mark position. |'a|
Jump       One position from the |jumplist|.
Change     One position from the |changelist|.

Data contained in the map:
Key  Type      Default  Description ~
l    UInteger  1        Position line number.  Must be
                        greater then zero.
c    UInteger  0        Position column number.
n    UInteger  34 ('"') Mark name.  Only valid for
                        GlobalMark and LocalMark
                        entries.
f    Binary    N/A      File name.  Required.
*    any       none     Other keys are allowed for
                        compatibility reasons, see
                        |shada-compatibility|.
```

- 9 (BufferList) Array containing maps. Each map in the array represents one buffer. Possible keys: Key Type Default Description

```
l    UInteger  1        Position line number.  Must be
                        greater then zero.
c    UInteger  0        Position column number.
f    Binary    N/A      File name.  Required.
*    any       none     Other keys are allowed for
                        compatibility reasons, see
                        |shada-compatibility|.
```

- * (Unknown) Any other entry type is allowed for compatibility reasons, see shada-compatibility.

E574 E575 E576 Errors in ShaDa file may have two types:

1. E575 for “logical” errors.
2. E576 for “critical” errors.

When writing, critical errors trigger behaviour described in shada-error-handling. When reading, critical errors cause the rest of the file to be skipped.

Critical errors include: shada-critical-contents-errors

- Any of first three MessagePack objects being not an unsigned integer.
- Third object requesting amount of bytes greater then bytes left in the ShaDa file.
- Entry with zero type. I.e. first object being equal to zero.
- MessagePack parser failing to parse the entry data.
- MessagePack parser consuming less or requesting greater bytes then described in the third object for parsing fourth object. I.e. when fourth object either contains more then one MessagePack object or it does not contain complete MessagePack object.

Standard Paths standard-path

Nvim stores configuration, data, and logs in standard locations. Plugins are strongly encouraged to follow this pattern also. Use stdpath() to get the paths.

base-directories xdg The "base" (root) directories conform to the XDG Base Directory Specification.
https://specifications.freedesktop.org/basedir/latest/

The $XDG_CONFIG_HOME, $XDG_DATA_HOME, $XDG_RUNTIME_DIR, $XDG_STATE_HOME, $XDG_CACHE_HOME, $XDG_CONFIG_DIRS and $XDG_DATA_DIRS environment variables are used if defined, else default values (listed below) are used.

Note: In the help these defaults are used as placeholders, e.g. "~/.config" is understood as "$XDG_CONFIG_HOME or ~/.config".

CONFIG DIRECTORY (DEFAULT)
$XDG_CONFIG_HOME Nvim: stdpath("config")

```
Unix:         ~/.config                  ~/.config/nvim
Windows:      ~/AppData/Local            ~/AppData/Local/nvim
```

DATA DIRECTORY (DEFAULT)
$XDG_DATA_HOME Nvim: stdpath("data")

```
Unix:         ~/.local/share             ~/.local/share/nvim
Windows:      ~/AppData/Local            ~/AppData/Local/nvim-data
```

RUN DIRECTORY (DEFAULT)
$XDG_RUNTIME_DIR Nvim: stdpath("run")

```
Unix:         /tmp/nvim.user/xxx         /tmp/nvim.user/xxx
Windows:      $TMP/nvim.user/xxx         $TMP/nvim.user/xxx
```

STATE DIRECTORY (DEFAULT)
$XDG_STATE_HOME Nvim: stdpath("state")

```
Unix:         ~/.local/state             ~/.local/state/nvim
Windows:      ~/AppData/Local            ~/AppData/Local/nvim-data
```

CACHE DIRECTORY (DEFAULT)
$XDG_CACHE_HOME Nvim: stdpath("cache")

```
Unix:         ~/.cache                   ~/.cache/nvim
Windows:      ~/AppData/Local/Temp       ~/AppData/Local/Temp/nvim-data
```

LOG DIRECTORY (DEFAULT)
$XDG_STATE_HOME/.../logs Nvim: stdpath("log")

```
Unix:         Not applicable             ~/.local/state/nvim/logs
Windows:      Not applicable             ~/AppData/Local/nvim-data/logs
```

LOG FILE (DEFAULT)
$NVIM_LOG_FILE Nvim: stdpath("log")/nvim.log

```
Unix:         ~/.local/state/nvim        ~/.local/state/nvim/logs/nvim.log
Windows:      ~/AppData/Local/nvim-data  ~/AppData/Local/nvim-data/logs/nvim.log
```

ADDITIONAL CONFIGS DIRECTORY (DEFAULT)
$XDG_CONFIG_DIRS Nvim: stdpath("config_dirs")

```
Unix:         /etc/xdg/                  /etc/xdg/nvim
Windows:      Not applicable             Not applicable
```

ADDITIONAL DATA DIRECTORY (DEFAULT)
$XDG_DATA_DIRS Nvim: stdpath("data_dirs")

```
Unix:         /usr/local/share           /usr/local/share/nvim
              /usr/share                 /usr/share/nvim
Windows:      Not applicable             Not applicable
```

NVIM_APPNAME **$NVIM_APPNAME**

The standard directories can be further configured by the $NVIM_APPNAME environment variable. This variable controls the sub-directory that Nvim will read from (and auto-create) in each of the base directories. For example, setting $NVIM_APPNAME to "foo" before starting will cause Nvim to look for configuration files in $XDG_CONFIG_HOME/foo instead of $XDG_CONFIG_HOME/nvim. $NVIM_APPNAME must be a name, such as "foo", or a relative path, such as "foo/bar".

Note: In the help wherever $XDG_CONFIG_…/nvim is mentioned it is understood as $XDG_CONFIG_…/$NVIM_APPNAME.

state-isolation

One use-case for $NVIM_APPNAME is to "isolate" Nvim applications. Alternatively, for true isolation, on Linux you can use cgroups namespaces:

```
systemd-run --user -qt -p PrivateUsers=yes -p BindPaths=/home/user/profile_xy:/home/user/.config/nvim nvim
```

stateless

To run Nvim without creating any directories or data files:

```
NVIM_LOG_FILE=/dev/null nvim -n -i NONE
```

LOG FILE **log** **$NVIM_LOG_FILE** **E5430**

Besides 'debug' and 'verbose', Nvim keeps a general log file for internal debugging, plugins and RPC clients. The log file can be viewed with:

```
:log nvim
```

Default location is stdpath("log")/nvim.log ($XDG_STATE_HOME/nvim/logs/nvim.log) unless that path is inaccessible or $NVIM_LOG_FILE was set before startup.

See log-files for details about other log files that can be created by Nvim during runtime or by plugins.

Supported platforms

supported-platforms

System	Tier	Versions	Tested versions
Linux (x86_64)	1	>= 2.6.32, glibc >= 2.12	Ubuntu 24.04
Linux (arm64)	1	>= 2.6.32, glibc >= 2.12	Ubuntu 24.04
macOS (x86_64)	1	>= 11	macOS 15
macOS (arm64)	1	>= 11	macOS 26
Windows 64-bit	1	>= Windows 10 Version 2004	Windows Server 2025
FreeBSD	2	>= 10	
OpenBSD	2	>= 7	
MinGW	2	MinGW-w64	
Windows 64-bit	3	< Windows 10 Version 2004	

Note: Windows 10 "Version 2004" or later is required for :terminal. To check your Windows version, run the "winver" command and look for "Version xxxx" (NOT "OS Build").

Note: On Windows "Server" you may need to install vcruntime140.dll: https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist?view=msvc-170

Support types

- Tier 1: Officially supported and tested with CI. Any contributed patch MUST NOT break support for such platforms.
- Tier 2: Officially supported, but not necessarily tested with CI. Support for these platforms are maintained by best effort, without being a top priority.
- Tier 3: Not tested and no guarantees, and not all features may work.

Adding support for a new platform

IMPORTANT: Before attempting to add support for a new platform please open an issue about it for discussion.

Common

Some common notes when adding support for new platforms:

CMake is the only supported build system. Nvim must be buildable on the platform with CMake.

All functionality related to the new platform must be implemented in its own file inside src/nvim/os unless it's already done in a common file, in which case adding an #ifdef is fine.

Syntax highlighting syntax-highlighting coloring

Syntax highlighting enables Vim to show parts of the text in another font or color. Those parts can be specific keywords or text matching a pattern. Vim doesn't parse the whole file (to keep it fast), so the highlighting has its limitations. Lexical highlighting might be a better name, but since everybody calls it syntax highlighting we'll stick with that.

Vim supports syntax highlighting on all terminals. But since most ordinary terminals have very limited highlighting possibilities, it works best in the GUI version, gvim.

In the User Manual:
usr_06.txt introduces syntax highlighting.
usr_44.txt introduces writing a syntax file.

1. Quick start :syn-qstart

:syn-enable :syntax-enable :syn-on :syntax-on
Syntax highlighting is enabled by default. If you need to enable it again after it was disabled (see below), use:

```
:syntax enable
```

Alternatively:

```
:syntax on
```

What this command actually does is to execute the command

```
:source $VIMRUNTIME/syntax/syntax.vim
```

If the VIM environment variable is not set, Vim will try to find the path in another way (see $VIMRUNTIME). Usually this works just fine. If it doesn't, try setting the VIM environment variable to the directory where the Vim stuff is located. For example, if your syntax files are in the "/usr/vim/vim82/syntax" directory, set $VIMRUNTIME to "/usr/vim/vim82". You must do this in the shell, before starting Vim. This command also sources the menu.vim script when the GUI is running or will start soon.

:hi-normal :highlight-normal
If you are running in the GUI, you can get white text on a black background with:

```
:highlight Normal guibg=Black guifg=White
```

For a color terminal see :hi-normal-cterm.

NOTE: The syntax files on MS-Windows have lines that end in <CR><NL>. The files for Unix end in <NL>. This means you should use the right type of file for your system. Although on MS-Windows the right format is automatically selected if the 'fileformats' option is not empty.

NOTE: When using reverse video ("gvim -fg white -bg black"), the default value of 'background' will not be set until the GUI window is opened, which is after reading the gvimrc. This will cause the wrong default highlighting to be used. To set the default value of 'background' before switching on highlighting, include the ":gui" command in the gvimrc:

```
:gui            " open window and set default for 'background'
:syntax on      " start highlighting, use 'background' to set colors
```

NOTE: Using ":gui" in the gvimrc means that "gvim -f" won't start in the foreground! Use ":gui -f" then.

g:syntax_on

You can toggle the syntax on/off with this command:

```
:if exists("g:syntax_on") | syntax off | else | syntax enable | endif
```

To put this into a mapping, you can use:

```
:map <F7> :if exists("g:syntax_on") <Bar>
     \   syntax off <Bar>
     \ else <Bar>
     \   syntax enable <Bar>
     \ endif <CR>
```

[using the <> notation, type this literally]

Details:
The ":syntax" commands are implemented by sourcing a file. To see exactly how this works, look in the file:
command file

```
    :syntax enable      $VIMRUNTIME/syntax/syntax.vim
    :syntax on          $VIMRUNTIME/syntax/syntax.vim
    :syntax manual      $VIMRUNTIME/syntax/manual.vim
    :syntax off         $VIMRUNTIME/syntax/nosyntax.vim
```

Also see syntax-loading.

NOTE: If displaying long lines is slow and switching off syntax highlighting makes it fast, consider setting the 'synmaxcol' option to a lower value.

2. Syntax files :syn-files

The syntax and highlighting commands for one language are normally stored in a syntax file. The name convention is: "{name}.vim". Where {name} is the name of the language, or an abbreviation (to fit the name in 8.3 characters, a requirement in case the file is used on a DOS filesystem).
Examples:

```
        c.vim           perl.vim        java.vim        html.vim
        cpp.vim         sh.vim          csh.vim
```

The syntax file can contain any Ex commands, just like a vimrc file. But the idea is that only commands for a specific language are included. When a language is a superset of another language, it may include the other one, for example, the cpp.vim file could include the c.vim file:

```
:so $VIMRUNTIME/syntax/c.vim
```

The .vim files are normally loaded with an autocommand. For example:

```
:au Syntax c        runtime! syntax/c.vim
:au Syntax cpp   runtime! syntax/cpp.vim
```

These commands are normally in the file $VIMRUNTIME/syntax/synload.vim.

MAKING YOUR OWN SYNTAX FILES mysyntaxfile

When you create your own syntax files, and you want to have Vim use these automatically with ":syntax enable", do this:

1. Create your user runtime directory. You would normally use the first item of the 'runtimepath' option. Example for Unix:

```
mkdir ~/.config/nvim
```

2. Create a directory in there called "syntax". For Unix:

```
mkdir ~/.config/nvim/syntax
```

3. Write the Vim syntax file. Or download one from the internet. Then write it in your syntax directory. For example, for the "mine" syntax:

```
:w ~/.config/nvim/syntax/mine.vim
```

Now you can start using your syntax file manually:

```
:set syntax=mine
```

You don't have to exit Vim to use this.

If you also want Vim to detect the type of file, see new-filetype.

If you are setting up a system with many users and you don't want each user to add the same syntax file, you can use another directory from 'runtimepath'.

ADDING TO AN EXISTING SYNTAX FILE mysyntaxfile-add

If you are mostly satisfied with an existing syntax file, but would like to add a few items or change the highlighting, follow these steps:

1. Create your user directory from 'runtimepath', see above.

2. Create a directory in there called "after/syntax". For Unix:

```
mkdir -p ~/.config/nvim/after/syntax
```

3. Write a Vim script that contains the commands you want to use. For example, to change the colors for the C syntax:

```
highlight cComment ctermfg=Green guifg=Green
```

4. Write that file in the "after/syntax" directory. Use the name of the

   syntax, with ".vim" added. For our C syntax:

```
:w ~/.config/nvim/after/syntax/c.vim
```

That's it. The next time you edit a C file the Comment color will be different. You don't even have to restart Vim.

If you have multiple files, you can use the filetype as the directory name. All the "*.vim" files in this directory will be used, for example:

```
~/.config/nvim/after/syntax/c/one.vim
~/.config/nvim/after/syntax/c/two.vim
```

REPLACING AN EXISTING SYNTAX FILE

mysyntaxfile-replace

If you don't like a distributed syntax file, or you have downloaded a new version, follow the same steps as for mysyntaxfile above. Just make sure that you write the syntax file in a directory that is early in 'runtimepath'. Vim will only load the first syntax file found, assuming that it sets b:current_syntax.

NAMING CONVENTIONS

group-name {group-name} E669 E5248

A syntax group name is to be used for syntax items that match the same kind of thing. These are then linked to a highlight group that specifies the color. A syntax group name doesn't specify any color or attributes itself.

The name for a highlight or syntax group must consist of ASCII letters, digits, underscores, dots, hyphens, or @. As a regexp: [a-zA-Z0-9_.@-]*. The maximum length of a group name is about 200 bytes. E1249

To be able to allow each user to pick their favorite set of colors, there must be preferred names for highlight groups that are common for many languages. These are the suggested group names (if syntax highlighting works properly you can see the actual color, except for "Ignore"):

```
Comment         any comment

Constant        any constant
String          a string constant: "this is a string"
Character       a character constant: 'c', '\n'
Number          a number constant: 234, 0xff
Boolean         a boolean constant: TRUE, false
Regexp          a regular expression constant: /ab*c/
Float           a floating point constant: 2.3e10

Identifier      any variable name
Function        function name (also: methods for classes)

Statement       any statement
Conditional     if, then, else, endif, switch, etc.
Repeat          for, do, while, etc.
Label           case, default, etc.
Operator        "sizeof", "+", "*", etc.
Keyword         any other keyword
Exception       try, catch, throw

PreProc         generic Preprocessor
Include         preprocessor #include
Define          preprocessor #define
Macro           same as Define
PreCondit       preprocessor #if, #else, #endif, etc.

Type            int, long, char, etc.
StorageClass    static, register, volatile, etc.
Structure       struct, union, enum, etc.
Typedef         a typedef

Special         any special symbol
SpecialChar     special character in a constant
Tag             you can use CTRL-] on this
Delimiter       character that needs attention
SpecialComment  special things inside a comment
Debug           debugging statements

Underlined      text that stands out, HTML links
Bold            bold text
Italic          italic text
BoldItalic      bold and italic text
Dimmed          text that is de-emphasized

Ignore          left blank, hidden  hl-Ignore

Error           any erroneous construct

Todo            anything that needs extra attention; mostly the
                keywords TODO FIXME and XXX

Added           added line in a diff
Changed         changed line in a diff
Removed         removed line in a diff
```

Note that highlight group names are not case sensitive. "String" and "string" can be used for the same group.

The following names are reserved and cannot be used as a group name:

```
NONE   ALL   ALLBUT   contains   contained
```

hl-Ignore

When using the Ignore group, you may also consider using the conceal
mechanism. See conceal.

3. Syntax loading procedure syntax-loading

This explains the details that happen when the command ":syntax enable" is
issued. When Vim initializes itself, it finds out where the runtime files are
located. This is used here as the variable $VIMRUNTIME.

":syntax enable" and ":syntax on" do the following:

```
    Source $VIMRUNTIME/syntax/syntax.vim
    |
    +-  Clear out any old syntax by sourcing $VIMRUNTIME/syntax/nosyntax.vim
    |
    +-  Source first syntax/synload.vim in 'runtimepath'
    |   |
    |   +-  Set up syntax autocmds to load the appropriate syntax file when
    |   |   the 'syntax' option is set. synload-1
    |   |
    |   +-  Source the user's optional file, from the mysyntaxfile variable.
    |       This is for backwards compatibility with Vim 5.x only. synload-2
    |
    +-  Do ":filetype on", which does ":runtime! filetype.vim".  It loads any
    |   filetype.vim files found.  It should always Source
    |   $VIMRUNTIME/filetype.vim, which does the following.
    |   |
    |   +-  Install autocmds based on suffix to set the 'filetype' option
    |   |   This is where the connection between file name and file type is
    |   |   made for known file types. synload-3
    |   |
    |   +-  Source the user's optional file, from the myfiletypefile
    |   |   variable.  This is for backwards compatibility with Vim 5.x only.
    |   |   synload-4
    |   |
    |   +-  Install one autocommand which sources scripts.vim when no file
    |   |   type was detected yet. synload-5
    |   |
    |   +-  Source $VIMRUNTIME/menu.vim, to setup the Syntax menu. menu.vim
    |
    +-  Install a FileType autocommand to set the 'syntax' option when a file
    |   type has been detected. synload-6
    |
    +-  Execute syntax autocommands to start syntax highlighting for each
        already loaded buffer.
```

Upon loading a file, Vim finds the relevant syntax file as follows:

```
    Loading the file triggers the BufReadPost autocommands.
    |
    +-  If there is a match with one of the autocommands from synload-3
    |   (known file types) or synload-4 (user's file types), the 'filetype'
    |   option is set to the file type.
    |
    +-  The autocommand at synload-5 is triggered.  If the file type was not
    |   found yet, then scripts.vim is searched for in 'runtimepath'.  This
    |   should always load $VIMRUNTIME/scripts.vim, which does the following.
    |   |
    |   +-  Source the user's optional file, from the myscriptsfile
    |   |   variable.  This is for backwards compatibility with Vim 5.x only.
    |   |
    |   +-  If the file type is still unknown, check the contents of the file,
    |       again with checks like "getline(1) =~ pattern" as to whether the
    |       file type can be recognized, and set 'filetype'.
    |
    +-  When the file type was determined and 'filetype' was set, this
    |   triggers the FileType autocommand synload-6 above.  It sets
    |   'syntax' to the determined file type.
    |
    +-  When the 'syntax' option was set above, this triggers an autocommand
    |   from synload-1 (and synload-2).  This find the main syntax file in
    |   'runtimepath', with this command:
    |           runtime! syntax/<name>.vim
    |
    +-  Any other user installed FileType or Syntax autocommands are
        triggered.  This can be used to change the highlighting for a specific
        syntax.
```

4. Conversion to HTML convert-to-HTML 2html.vim

The old to html converter has been replaced by a Lua version and the
documentation has been moved to :TOhtml.

5. Syntax file remarks :syn-file-remarks

b:current_syntax-variable

Vim stores the name of the syntax that has been loaded in the

"b:current_syntax" variable. You can use this if you want to load other
settings, depending on which syntax is active. Example:

```
:au BufReadPost * if b:current_syntax == "csh"
:au BufReadPost *   do-some-things
:au BufReadPost * endif
```

ABEL **ft-abel-syntax**

ABEL highlighting provides some user-defined options. To enable them, assign
any value to the respective variable. Example:

```
:let abel_obsolete_ok=1
```

To disable them use ":unlet". Example:

```
:unlet abel_obsolete_ok
```

Variable	Highlight
abel_obsolete_ok	obsolete keywords are statements, not errors
abel_cpp_comments_illegal	do not interpret '//' as inline comment leader

ADA

See ft-ada-syntax

ALGOL 68 **algol68** **ft-algol68-syntax**

This syntax file currently targets the Algol 68 Genie project using the
default UPPER stropping regime. It should also be usable in other
environments using the UPPER stropping regime, though somewhat less complete.

Highlighting can be further configured with the following variables.

Variable	Highlight
algol68_no_preludes	no prelude identifiers, procedures or operators

ANT **ft-ant-syntax**

The ant syntax file provides syntax highlighting for javascript and python
by default. Syntax highlighting for other script languages can be installed
by the function AntSyntaxScript(), which takes the tag name as first argument
and the script syntax file name as second argument. Example:

```
:call AntSyntaxScript('perl', 'perl.vim')
```

will install syntax perl highlighting for the following ant code

```
<script language = 'perl'><![CDATA[
    # everything inside is highlighted as perl
]]></script>
```

See mysyntaxfile-add for installing script languages permanently.

APACHE **ft-apache-syntax**

The apache syntax file provides syntax highlighting for Apache HTTP server
version 2.2.3.

ASSEMBLY **asm68k** **ft-asm-syntax** **ft-asmh8300-syntax** **ft-nasm-syntax**

ft-masm-syntax ft-asm68k-syntax

Files matching "*.i" could be Progress or Assembly. If the automatic
detection doesn't work for you, or you don't edit Progress at all, use this in
your startup vimrc:

```
:let filetype_i = "asm"
```

Replace "asm" with the type of assembly you use.

There are many types of assembly languages that all use the same file name
extensions. Therefore you will have to select the type yourself, or add a
line in the assembly file that Vim will recognize. Currently these syntax
files are included:

```
        asm             GNU assembly (usually have .s or .S extension and were
                        already built using C compiler such as GCC or CLANG)
        asm68k          Motorola 680x0 assembly
        asmh8300        Hitachi H-8300 version of GNU assembly
        ia64            Intel Itanium 64
        fasm            Flat assembly (https://flatassembler.net)
        masm            Microsoft assembly (.masm files are compiled with
                        Microsoft's Macro Assembler.  This is only supported
                        for x86, x86_64, ARM and AARCH64 CPU families)
        nasm            Netwide assembly
```

```
        tasm            Turbo Assembly (with opcodes 80x86 up to Pentium, and
                        MMX)
        pic             PIC assembly (currently for PIC16F84)
```

The most flexible is to add a line in your assembly file containing:

```
asmsyntax=nasm
```

Replace "nasm" with the name of the real assembly syntax. This line must be one of the first five lines in the file. No non-white text must be immediately before or after this text. Note that specifying asmsyntax=foo is equivalent to setting ft=foo in a modeline, and that in case of a conflict between the two settings the one from the modeline will take precedence (in particular, if you have ft=asm in the modeline, you will get the GNU syntax highlighting regardless of what is specified as asmsyntax).

The syntax type can always be overruled for a specific buffer by setting the b:asmsyntax variable:

```
:let b:asmsyntax = "nasm"
```

If b:asmsyntax is not set, either automatically or by hand, then the value of the global variable asmsyntax is used. This can be seen as a default assembly language:

```
:let asmsyntax = "nasm"
```

As a last resort, if nothing is defined, the "asm" syntax is used.

Netwide assembler (nasm.vim) optional highlighting

To enable a feature:

```
:let   {variable}=1|set syntax=nasm
```

To disable a feature:

```
:unlet {variable}  |set syntax=nasm
```

Variable	**Highlight**
nasm_loose_syntax	unofficial parser allowed syntax not as Error (parser dependent; not recommended)
nasm_ctx_outside_macro	contexts outside macro not as Error
nasm_no_warn	potentially risky syntax not as ToDo

ASTRO ft-astro-syntax

Configuration

The following variables control certain syntax highlighting features.
You can add them to your .vimrc.

To enable TypeScript and TSX for ".astro" files (default "disable"):

```
let g:astro_typescript = "enable"
```

To enable Stylus for ".astro" files (default "disable"):

```
let g:astro_stylus = "enable"
```

NOTE: You need to install an external plugin to support stylus in astro files.

ASPPERL ft-aspperl-syntax

ASPVBS ft-aspvbs-syntax

*.asp and *.asa files could be either Perl or Visual Basic script. Since it's hard to detect this you can set two global variables to tell Vim what you are using. For Perl script use:

```
:let g:filetype_asa = "aspperl"
:let g:filetype_asp = "aspperl"
```

For Visual Basic use:

```
:let g:filetype_asa = "aspvbs"
:let g:filetype_asp = "aspvbs"
```

ASYMPTOTE ft-asy-syntax

By default, only basic Asymptote keywords are highlighted. To highlight extended geometry keywords:

```
:let g:asy_syn_plain = 1
```

and for highlighting keywords related to 3D constructions:

```
:let g:asy_syn_three = 1
```

By default, Asymptote-defined colors (e.g: lightblue) are highlighted. To highlight TeX-defined colors (e.g: BlueViolet) use:

```
:let g:asy_syn_texcolors = 1
```

or for Xorg colors (e.g: AliceBlue):

```
:let g:asy_syn_x11colors = 1
```

BAAN **baan-syntax**

The baan.vim gives syntax support for BaanC of release BaanIV up to SSA ERP LN for both 3 GL and 4 GL programming. Large number of standard defines/constants are supported.

Some special violation of coding standards will be signalled when one specify in ones init.vim:

```
let baan_code_stds=1
```

baan-folding

Syntax folding can be enabled at various levels through the variables mentioned below (Set those in your init.vim). The more complex folding on source blocks and SQL can be CPU intensive.

To allow any folding and enable folding at function level use:

```
let baan_fold=1
```

Folding can be enabled at source block level as if, while, for ,... The indentation preceding the begin/end keywords has to match (spaces are not considered equal to a tab).

```
let baan_fold_block=1
```

Folding can be enabled for embedded SQL blocks as SELECT, SELECTDO, SELECTEMPTY, ... The indentation preceding the begin/end keywords has to match (spaces are not considered equal to a tab).

```
let baan_fold_sql=1
```

Note: Block folding can result in many small folds. It is suggested to :set the options 'foldminlines' and 'foldnestmax' in init.vim or use :setlocal in .../after/syntax/baan.vim (see after-directory). Eg:

```
set foldminlines=5
set foldnestmax=6
```

BASIC **ft-basic-syntax** **ft-vb-syntax**

Both Visual Basic and "normal" BASIC use the extension ".bas". To detect which one should be used, Vim checks for the string "VB_Name" in the first five lines of the file. If it is not found, filetype will be "basic", otherwise "vb". Files with the ".frm" extension will always be seen as Visual Basic.

If the automatic detection doesn't work for you or you only edit, for example, FreeBASIC files, use this in your startup vimrc:

```
:let filetype_bas = "freebasic"
```

C ft-c-syntax

A few things in C highlighting are optional. To enable them assign any value (including zero) to the respective variable. Example:

```
:let c_comment_strings = 1
:let c_no_bracket_error = 0
```

To disable them use :unlet. Example:

```
:unlet c_comment_strings
```

Setting the value to zero doesn't work!

An alternative is to switch to the C++ highlighting:

```
:set filetype=cpp
```

Variable	**Highlight**
c_gnu	GNU gcc specific items
c_comment_strings	strings and numbers inside a comment
c_space_errors	trailing white space and spaces before a <Tab>
c_no_trail_space_error	... but no trailing spaces
c_no_tab_space_error	... but no spaces before a <Tab>
c_no_bracket_error	don't highlight {}; inside [] as errors
c_no_curly_error	don't highlight {}; inside [] and () as errors; ...except { and } in first column Default is to highlight them, otherwise you can't spot a missing ")".
c_curly_error	highlight a missing } by finding all pairs; this forces syncing from the start of the file, can be slow
c_no_ansi	don't do standard ANSI types and constants
c_ansi_typedefs	... but do standard ANSI types
c_ansi_constants	... but do standard ANSI constants
c_no_utf	don't highlight \u and \U in strings
c_syntax_for_h	use C syntax for *.h files instead of C++/ObjC/ObjC++ (**NOTE:** This variable is deprecated and no longer necessary, as *.h files now default to C, unless the file contains C++ or Objective-C syntax. If the automated detection fails, the default filetype can be adjusted using g:filetype_h.)
c_no_if0	don't highlight "#if 0" blocks as comments
c_no_cformat	don't highlight %-formats in strings
c_no_c99	don't highlight C99 standard items
c_no_c11	don't highlight C11 standard items
c_no_c23	don't highlight C23 standard items
c_no_c29	don't highlight C29 standard items
c_no_bsd	don't highlight BSD specific types
c_functions	highlight function calls and definitions
c_function_pointers	highlight function pointers definitions

When 'foldmethod' is set to "syntax" then /* */ comments and { } blocks will become a fold. If you don't want comments to become a fold use:

```
:let c_no_comment_fold = 1
```

"#if 0" blocks are also folded, unless:

```
:let c_no_if0_fold = 1
```

If you notice highlighting errors while scrolling backwards, which are fixed when redrawing with CTRL-L, try setting the "c_minlines" internal variable to a larger number:

```
:let c_minlines = 100
```

This will make the syntax synchronization start 100 lines before the first displayed line. The default value is 50 (15 when c_no_if0 is set). The disadvantage of using a larger number is that redrawing can become slow.

When using the "#if 0" / "#endif" comment highlighting, notice that this only works when the "#if 0" is within "c_minlines" from the top of the window. If you have a long "#if 0" construct it will not be highlighted correctly.

To match extra items in comments, use the cCommentGroup cluster.
Example:

```
:au Syntax c call MyCadd()
:function MyCadd()
:  syn keyword cMyItem contained Ni
:  syn cluster cCommentGroup add=cMyItem
:  hi link cMyItem Title
:endfun
```

ANSI constants will be highlighted with the "cConstant" group. This includes "NULL", "SIG_IGN" and others. But not "TRUE", for example, because this is not in the ANSI standard. If you find this confusing, remove the cConstant highlighting:

```
:hi link cConstant NONE
```

If you see '{' and '}' highlighted as an error where they are OK, reset the highlighting for cErrInParen and cErrInBracket.

If you want to use folding in your C files, you can add these lines in a file in the "after" directory in 'runtimepath'. For Unix this would be ~/.config/nvim/after/syntax/c.vim.

```
syn sync fromstart
set foldmethod=syntax
```

CANGJIE

ft-cangjie-syntax

This file provides syntax highlighting for the Cangjie programming language, a new-generation language oriented to full-scenario intelligence.

All highlighting is enabled by default. To disable highlighting for a specific group, set the corresponding variable to 0 in your vimrc.
All options to disable highlighting are:

```
:let g:cangjie_builtin_color = 0
:let g:cangjie_comment_color = 0
:let g:cangjie_identifier_color = 0
:let g:cangjie_keyword_color = 0
:let g:cangjie_macro_color = 0
:let g:cangjie_number_color = 0
:let g:cangjie_operator_color = 0
:let g:cangjie_string_color = 0
:let g:cangjie_type_color = 0
```

CH

ft-ch-syntax

C/C++ interpreter. Ch has similar syntax highlighting to C and builds upon the C syntax file. See ft-c-syntax for all the settings that are available for C.

By setting a variable you can tell Vim to use Ch syntax for *.h files, instead of C or C++:

```
:let g:filetype_h = 'ch'
```

CHILL

ft-chill-syntax

Chill syntax highlighting is similar to C. See ft-c-syntax for all the settings that are available. Additionally there is:

```
chill_space_errors      like c_space_errors
chill_comment_string    like c_comment_strings
chill_minlines          like c_minlines
```

CHANGELOG

ft-changelog-syntax

ChangeLog supports highlighting spaces at the start of a line.
If you do not like this, add following line to your vimrc:

```
let g:changelog_spacing_errors = 0
```

This works the next time you edit a changelog file. You can also use "b:changelog_spacing_errors" to set this per buffer (before loading the syntax file).

You can change the highlighting used, e.g., to flag the spaces as an error:

```
:hi link ChangelogError Error
```

Or to avoid the highlighting:

```
:hi link ChangelogError NONE
```

This works immediately.

CLOJURE

ft-clojure-syntax

g:clojure_syntax_keywords

Syntax highlighting of public vars in "clojure.core" is provided by default, but additional symbols can be highlighted by adding them to the g:clojure_syntax_keywords variable. The value should be a Dictionary of syntax group names, each containing a List of identifiers.

```
let g:clojure_syntax_keywords = {
    \   'clojureMacro': ["defproject", "defcustom"],
    \   'clojureFunc': ["string/join", "string/replace"]
    \ }
```

Refer to the Clojure syntax script for valid syntax group names.

There is also b:clojure_syntax_keywords which is a buffer-local variant of this variable intended for use by plugin authors to highlight symbols dynamically.

By setting the b:clojure_syntax_without_core_keywords variable, vars from "clojure.core" will not be highlighted by default. This is useful for namespaces that have set (:refer-clojure :only [])

g:clojure_fold

Setting g:clojure_fold to 1 will enable the folding of Clojure code. Any list, vector or map that extends over more than one line can be folded using the standard Vim fold-commands.

g:clojure_discard_macro

Set this variable to 1 to enable basic highlighting of Clojure's "discard reader macro".

```
#_(defn foo [x]
    (println x))
```

Note that this option will not correctly highlight stacked discard macros (e.g. #_#_).

COBOL **ft-cobol-syntax**

COBOL highlighting has different needs for legacy code than it does for fresh development. This is due to differences in what is being done (maintenance versus development) and other factors. To enable legacy code highlighting, add this line to your vimrc:

```
:let cobol_legacy_code = 1
```

To disable it again, use this:

```
:unlet cobol_legacy_code
```

COLD FUSION **ft-coldfusion-syntax**

The ColdFusion has its own version of HTML comments. To turn on ColdFusion comment highlighting, add the following line to your startup file:

```
:let html_wrong_comments = 1
```

The ColdFusion syntax file is based on the HTML syntax file.

CPP **ft-cpp-syntax**

Most things are the same as ft-c-syntax.

Variable	Highlight
cpp_no_cpp11	don't highlight C++11 standard items
cpp_no_cpp14	don't highlight C++14 standard items
cpp_no_cpp17	don't highlight C++17 standard items
cpp_no_cpp20	don't highlight C++20 standard items
cpp_no_cpp23	don't highlight C++23 standard items
cpp_no_cpp26	don't highlight C++26 standard items

CSH **ft-csh-syntax**

This covers the shell named "csh". Note that on some systems tcsh is actually used.

Detecting whether a file is csh or tcsh is notoriously hard. Some systems symlink /bin/csh to /bin/tcsh, making it almost impossible to distinguish between csh and tcsh. In case VIM guesses wrong you can set the "filetype_csh" variable. For using csh: g:filetype_csh

```
:let g:filetype_csh = "csh"
```

For using tcsh:

```
:let g:filetype_csh = "tcsh"
```

Any script with a tcsh extension or a standard tcsh filename (.tcshrc, tcsh.tcshrc, tcsh.login) will have filetype tcsh. All other tcsh/csh scripts will be classified as tcsh, UNLESS the "filetype_csh" variable exists. If the "filetype_csh" variable exists, the filetype will be set to the value of the variable.

CSV **ft-csv-syntax**

If you change the delimiter of a CSV file, its syntax highlighting will no longer match the changed file content. You will need to unlet the following variable:

```
:unlet b:csv_delimiter
```

And afterwards save and reload the file:

```
:w
:e
```

Now the syntax engine should determine the newly changed CSV delimiter.

CYNLIB ft-cynlib-syntax

Cynlib files are C++ files that use the Cynlib class library to enable
hardware modelling and simulation using C++. Typically Cynlib files have a
.cc or a .cpp extension, which makes it very difficult to distinguish them
from a normal C++ file. Thus, to enable Cynlib highlighting for .cc files,
add this line to your vimrc file:

```
:let cynlib_cyntax_for_cc=1
```

Similarly for cpp files (this extension is only usually used in Windows)

```
:let cynlib_cyntax_for_cpp=1
```

To disable these again, use this:

```
:unlet cynlib_cyntax_for_cc
:unlet cynlib_cyntax_for_cpp
```

CWEB ft-cweb-syntax

Files matching "*.w" could be Progress or cweb. If the automatic detection
doesn't work for you, or you don't edit Progress at all, use this in your
startup vimrc:

```
:let filetype_w = "cweb"
```

CSHARP ft-cs-syntax

C# raw string literals may use any number of quote marks to encapsulate the
block, and raw interpolated string literals may use any number of braces to
encapsulate the interpolation, e.g.

```
$$$"""Hello {{{name}}}"""
```

By default, Vim highlights 3-8 quote marks, and 1-8 interpolation braces.
The maximum numbers of quotes and braces recognized can configured using the
following variables:

Variable	Default
g:cs_raw_string_quote_count	8
g:cs_raw_string_interpolation_brace_count	8

DART ft-dart-syntax

Dart is an object-oriented, typed, class defined, garbage collected language
used for developing mobile, desktop, web, and back-end applications. Dart
uses a C-like syntax derived from C, Java, and JavaScript, with features
adopted from Smalltalk, Python, Ruby, and others.

More information about the language and its development environment at the
official Dart language website at https://dart.dev

dart.vim syntax detects and highlights Dart statements, reserved words,
type declarations, storage classes, conditionals, loops, interpolated values,
and comments. There is no support idioms from Flutter or any other Dart
framework.

Changes, fixes? Submit an issue or pull request via:

https://github.com/pr3d4t0r/dart-vim-syntax/

DESKTOP ft-desktop-syntax

Primary goal of this syntax file is to highlight .desktop and .directory files
according to freedesktop.org standard:
https://specifications.freedesktop.org/desktop-entry-spec/latest/
To highlight nonstandard extensions that does not begin with X-, set

```
let g:desktop_enable_nonstd = 1
```

Note that this may cause wrong highlight.
To highlight KDE-reserved features, set

```
let g:desktop_enable_kde = 1
```

g:desktop_enable_kde follows g:desktop_enable_nonstd if not supplied

DIFF diff.vim

The diff highlighting normally finds translated headers. This can be slow if
there are very long lines in the file. To disable translations:

```
:let diff_translations = 0
```

Also see diff-slow.

DIRCOLORS ft-dircolors-syntax

The dircolors utility highlighting definition has one option. It exists to
provide compatibility with the Slackware GNU/Linux distributions version of
the command. It adds a few keywords that are generally ignored by most
versions. On Slackware systems, however, the utility accepts the keywords and
uses them for processing. To enable the Slackware keywords add the following
line to your startup file:

```
let dircolors_is_slackware = 1
```

DOCBOOK ft-docbk-syntax docbook

DOCBOOK XML ft-docbkxml-syntax

DOCBOOK SGML ft-docbksgml-syntax

There are two types of DocBook files: SGML and XML. To specify what type you
are using the "b:docbk_type" variable should be set. Vim does this for you
automatically if it can recognize the type. When Vim can't guess it the type
defaults to XML.
You can set the type manually:

```
:let b:docbk_type = "sgml"
```

or:

```
:let b:docbk_type = "xml"
```

You need to do this before loading the syntax file, which is complicated.
Simpler is setting the filetype to "docbkxml" or "docbksgml":

```
:set filetype=docbksgml
```

or:

```
:set filetype=docbkxml
```

You can specify the DocBook version for the current buffer:

```
:let b:docbk_ver = 3
```

When "b:docbk_ver" is not set, "g:docbk_ver" is used as a global fallback:

```
:let g:docbk_ver = 3
```

When neither variable is set, version 4 is used.

DOSBATCH ft-dosbatch-syntax

Select the set of Windows Command interpreter extensions that should be
supported with the variable dosbatch_cmdextversion. For versions of Windows
NT (before Windows 2000) this should have the value of 1. For Windows 2000
and later it should be 2.
Select the version you want with the following line:

```
:let dosbatch_cmdextversion = 1
```

If this variable is not defined it defaults to a value of 2 to support
Windows 2000 and later.

The original MS-DOS supports an idiom of using a double colon (::) as an
alternative way to enter a comment line. This idiom can be used with the
current Windows Command Interpreter, but it can lead to problems when used
inside (...) command blocks. You can find a discussion about this on
Stack Overflow -

https://stackoverflow.com/questions/12407800/which-comment-style-should-i-use-in-batch-files

To allow the use of the :: idiom for comments in command blocks with the
Windows Command Interpreter set the dosbatch_colons_comment variable to
anything:

```
:let dosbatch_colons_comment = 1
```

If this variable is set then a :: comment that is the last line in a command
block will be highlighted as an error.

There is an option that covers whether *.btm files should be detected as type
"dosbatch" (MS-DOS batch files) or type "btm" (4DOS batch files). The latter
is used by default. You may select the former with the following line:

```
:let g:dosbatch_syntax_for_btm = 1
```

If this variable is undefined or zero, btm syntax is selected.

DOXYGEN doxygen-syntax

Doxygen generates code documentation using a special documentation format
(similar to Javadoc). This syntax script adds doxygen highlighting to c, cpp,
idl and php files, and should also work with java.

There are a few of ways to turn on doxygen formatting. It can be done
explicitly or in a modeline by appending '.doxygen' to the syntax of the file.
Example:

```
:set syntax=c.doxygen
```

or

```
// vim:syntax=c.doxygen
```

It can also be done automatically for C, C++, C#, IDL and PHP files by setting
the global or buffer-local variable load_doxygen_syntax. This is done by
adding the following to your vimrc.

```
:let g:load_doxygen_syntax=1
```

There are a couple of variables that have an effect on syntax highlighting,
and are to do with non-standard highlighting options.

Variable	Default	Effect
g:doxygen_enhanced_color		
g:doxygen_enhanced_colour	0	Use non-standard highlighting for doxygen comments.
doxygen_my_rendering	0	Disable rendering of HTML bold, italic and html_my_rendering underline.
doxygen_javadoc_autobrief	1	Set to 0 to disable javadoc autobrief colour highlighting.
doxygen_end_punctuation	'[.]'	Set to regexp match for the ending punctuation of brief

There are also some highlight groups worth mentioning as they can be useful in
configuration.

Highlight	Effect
doxygenErrorComment	The colour of an end-comment when missing punctuation in a code, verbatim or dot section
doxygenLinkError	The colour of an end-comment when missing the \endlink from a \link section.

DTD ft-dtd-syntax

The DTD syntax highlighting is case sensitive by default. To disable
case-sensitive highlighting, add the following line to your startup file:

```
:let dtd_ignore_case=1
```

The DTD syntax file will highlight unknown tags as errors. If
this is annoying, it can be turned off by setting:

```
:let dtd_no_tag_errors=1
```

before sourcing the dtd.vim syntax file.
Parameter entity names are highlighted in the definition using the
'Type' highlighting group and 'Comment' for punctuation and '%'.
Parameter entity instances are highlighted using the 'Constant'
highlighting group and the 'Type' highlighting group for the
delimiters % and ;. This can be turned off by setting:

```
:let dtd_no_param_entities=1
```

The DTD syntax file is also included by xml.vim to highlight included dtd's.

EIFFEL

ft-eiffel-syntax

While Eiffel is not case-sensitive, its style guidelines are, and the
syntax highlighting file encourages their use. This also allows to
highlight class names differently. If you want to disable case-sensitive
highlighting, add the following line to your startup file:

```
:let eiffel_ignore_case=1
```

Case still matters for class names and TODO marks in comments.

Conversely, for even stricter checks, add one of the following lines:

```
:let eiffel_strict=1
:let eiffel_pedantic=1
```

Setting eiffel_strict will only catch improper capitalization for the
five predefined words "Current", "Void", "Result", "Precursor", and
"NONE", to warn against their accidental use as feature or class names.

Setting eiffel_pedantic will enforce adherence to the Eiffel style
guidelines fairly rigorously (like arbitrary mixes of upper- and
lowercase letters as well as outdated ways to capitalize keywords).

If you want to use the lower-case version of "Current", "Void",
"Result", and "Precursor", you can use

```
:let eiffel_lower_case_predef=1
```

instead of completely turning case-sensitive highlighting off.

Support for ISE's proposed new creation syntax that is already
experimentally handled by some compilers can be enabled by:

```
:let eiffel_ise=1
```

Finally, some vendors support hexadecimal constants. To handle them, add

```
:let eiffel_hex_constants=1
```

to your startup file.

EUPHORIA

ft-euphoria-syntax

Two syntax highlighting files exist for Euphoria. One for Euphoria
version 3.1.1, which is the default syntax highlighting file, and one for
Euphoria version 4.0.5 or later.

Euphoria version 3.1.1 (https://www.rapideuphoria.com/ link seems dead) is
still necessary for developing applications for the DOS platform, which
Euphoria version 4 (https://www.openeuphoria.org/) does not support.

The following file extensions are auto-detected as Euphoria file type:

```
*.e, *.eu, *.ew, *.ex, *.exu, *.exw
*.E, *.EU, *.EW, *.EX, *.EXU, *.EXW
```

To select syntax highlighting file for Euphoria, as well as for
auto-detecting the *.e and *.E file extensions as Euphoria file type,
add the following line to your startup file:

```
:let g:filetype_euphoria = "euphoria3"
```

       or

```
:let g:filetype_euphoria = "euphoria4"
```

Elixir and Euphoria share the *.ex file extension. If the filetype is
specifically set as Euphoria with the g:filetype_euphoria variable, or the
file is determined to be Euphoria based on keywords in the file, then the
filetype will be set as Euphoria. Otherwise, the filetype will default to
Elixir.

ERLANG

ft-erlang-syntax

Erlang is a functional programming language developed by Ericsson. Files with
the following extensions are recognized as Erlang files: erl, hrl, yaws.

Vim highlights triple-quoted docstrings as comments by default.

If you want triple-quoted docstrings highlighted as Markdown, add the
following line to your vimrc:

```
:let g:erlang_use_markdown_for_docs = 1
```

The plain text inside the docstrings (that is, the characters that are not
highlighted by the Markdown syntax) is still highlighted as a comment.

If you want to highlight the plain text inside the docstrings using a different highlight group, add the following line to your vimrc (the example highlights plain text using the String highlight group):

```
:let g:erlang_docstring_default_highlight = 'String'
```

If you don't enable Markdown, this line highlights the full docstrings according to the specified highlight group.

Use the following line to disable highlighting for the plain text:

```
:let g:erlang_docstring_default_highlight = ''
```

Configuration examples:

```
" Highlight docstrings as Markdown.
:let g:erlang_use_markdown_for_docs = 1

" 1. Highlight Markdown elements in docstrings as Markdown.
" 2. Highlight the plain text in docstrings as String.
:let g:erlang_use_markdown_for_docs = 1
:let g:erlang_docstring_default_highlight = 'String'

" Highlight docstrings as strings (no Markdown).
:let g:erlang_docstring_default_highlight = 'String'

" 1. Highlight Markdown elements in docstrings as Markdown.
" 2. Don't highlight the plain text in docstrings.
:let g:erlang_use_markdown_for_docs = 1
:let g:erlang_docstring_default_highlight = ''
```

ELIXIR ft-elixir-syntax

Elixir is a dynamic, functional language for building scalable and maintainable applications.

The following file extensions are auto-detected as Elixir file types:

```
*.ex, *.exs, *.eex, *.leex, *.lock
```

Elixir and Euphoria share the *.ex file extension. If the filetype is specifically set as Euphoria with the g:filetype_euphoria variable, or the file is determined to be Euphoria based on keywords in the file, then the filetype will be set as Euphoria. Otherwise, the filetype will default to Elixir.

FLEXWIKI ft-flexwiki-syntax

FlexWiki is an ASP.NET-based wiki package available at www.flexwiki.com
NOTE: This site currently doesn't work, on Wikipedia is mentioned that development stopped in 2009.

Syntax highlighting is available for the most common elements of FlexWiki syntax. The associated ftplugin script sets some buffer-local options to make editing FlexWiki pages more convenient. FlexWiki considers a newline as the start of a new paragraph, so the ftplugin sets 'tw'=0 (unlimited line length), 'wrap' (wrap long lines instead of using horizontal scrolling), 'linebreak' (to wrap at a character in 'breakat' instead of at the last char on screen), and so on. It also includes some keymaps that are disabled by default.

If you want to enable the keymaps that make "j" and "k" and the cursor keys move up and down by display lines, add this to your vimrc:

```
:let flexwiki_maps = 1
```

FORM ft-form-syntax

The coloring scheme for syntax elements in the FORM file uses the default modes Conditional, Number, Statement, Comment, PreProc, Type, and String, following the language specifications in 'Symbolic Manipulation with FORM' by J.A.M. Vermaseren, CAN, Netherlands, 1991.

If you want to include your own changes to the default colors, you have to redefine the following syntax groups:

- formConditional
- formNumber
- formStatement
- formHeaderStatement

- formComment
- formPreProc
- formDirective
- formType
- formString

Note that the form.vim syntax file implements FORM preprocessor commands and directives per default in the same syntax group.

A predefined enhanced color mode for FORM is available to distinguish between header statements and statements in the body of a FORM program. To activate this mode define the following variable in your vimrc file

```
:let form_enhanced_color=1
```

The enhanced mode also takes advantage of additional color features for a dark gvim display. Here, statements are colored LightYellow instead of Yellow, and conditionals are LightBlue for better distinction.

Both Visual Basic and FORM use the extension ".frm". To detect which one should be used, Vim checks for the string "VB_Name" in the first five lines of the file. If it is found, filetype will be "vb", otherwise "form".

If the automatic detection doesn't work for you or you only edit, for example, FORM files, use this in your startup vimrc:

```
:let filetype_frm = "form"
```

FORTH ft-forth-syntax

Files matching "*.f" could be Fortran or Forth and those matching "*.fs" could be F# or Forth. If the automatic detection doesn't work for you, or you don't edit F# or Fortran at all, use this in your startup vimrc:

```
:let filetype_f  = "forth"
:let filetype_fs = "forth"
```

FORTRAN ft-fortran-syntax

Default highlighting and dialect
Vim highlights according to Fortran 2023 (the most recent standard). This choice should be appropriate for most users most of the time because Fortran 2023 is almost a superset of previous versions (Fortran 2018, 2008, 2003, 95, 90, 77, and 66). A few legacy constructs deleted or declared obsolescent, respectively, in recent Fortran standards are highlighted as errors and todo items.

The syntax script no longer supports Fortran dialects. The variable fortran_dialect is now silently ignored. Since computers are much faster now, the variable fortran_more_precise is no longer needed and is silently ignored.

Fortran source code form
Fortran code can be in either fixed or free source form. Note that the syntax highlighting will not be correct if the form is incorrectly set.

When you create a new Fortran file, the syntax script assumes fixed source form. If you always use free source form, then

```
:let fortran_free_source=1
```

If you always use fixed source form, then

```
:let fortran_fixed_source=1
```

If the form of the source code depends, in a non-standard way, upon the file extension, then it is most convenient to set fortran_free_source in a ftplugin file. For more information on ftplugin files, see ftplugin. Note that this will work only if the "filetype plugin indent on" command precedes the "syntax on" command in your .vimrc file.

When you edit an existing Fortran file, the syntax script will assume free source form if the fortran_free_source variable has been set, and assumes fixed source form if the fortran_fixed_source variable has been set. Suppose neither of these variables have been set. In that case, the syntax script attempts to determine which source form has been used by examining the file extension using conventions common to the ifort, gfortran, Cray, NAG, and PathScale compilers (.f, .for, .f77 for fixed-source, .f90, .f95, .f03, .f08 for free-source). No default is used for the .fpp and .ftn file extensions

because different compilers treat them differently. If none of this works, then the script examines the first five columns of the first 500 lines of your file. If no signs of free source form are detected, then the file is assumed to be in fixed source form. The algorithm should work in the vast majority of cases. In some cases, such as a file that begins with 500 or more full-line comments, the script may incorrectly decide that the code is in fixed form. If that happens, just add a non-comment statement beginning anywhere in the first five columns of the first twenty-five lines, save (:w), and then reload (:e!) the file.

Vendor extensions
Fixed-form Fortran requires a maximum line length of 72 characters but the script allows a maximum line length of 80 characters as do all compilers created in the last three decades. An even longer line length of 132 characters is allowed if you set the variable fortran_extended_line_length with a command such as

```
:let fortran_extended_line_length=1
```

If you want additional highlighting of the CUDA Fortran extensions, you should set the variable fortran_CUDA with a command such as

```
:let fortran_CUDA=1
```

To activate recognition of some common, non-standard, vendor-supplied intrinsics, you should set the variable fortran_vendor_intrinsics with a command such as

```
:let fortran_vendor_intrinsics=1
```

Tabs in Fortran files
Tabs are not recognized by the Fortran standards. Tabs are not a good idea in fixed format Fortran source code which requires fixed column boundaries. Therefore, tabs are marked as errors. Nevertheless, some programmers like using tabs. If your Fortran files contain tabs, then you should set the variable fortran_have_tabs in your vimrc with a command such as

```
:let fortran_have_tabs=1
```

Unfortunately, the use of tabs will mean that the syntax file will not be able to detect incorrect margins.

Syntax folding of Fortran files
Vim will fold your file using foldmethod=syntax, if you set the variable fortran_fold in your .vimrc with a command such as

```
:let fortran_fold=1
```

to instruct the syntax script to define fold regions for program units, that is main programs starting with a program statement, subroutines, function subprograms, modules, submodules, blocks of comment lines, and block data units. Block, interface, associate, critical, type definition, and change team constructs will also be folded. If you also set the variable fortran_fold_conditionals with a command such as

```
:let fortran_fold_conditionals=1
```

then fold regions will also be defined for do loops, if blocks, select case, select type, and select rank constructs. Note that defining fold regions can be slow for large files.

The syntax/fortran.vim script contains embedded comments that tell you how to comment and/or uncomment some lines to (a) activate recognition of some non-standard, vendor-supplied intrinsics and (b) to prevent features deleted or declared obsolescent in the 2008 standard from being highlighted as todo items.

Limitations
Parenthesis checking does not catch too few closing parentheses. Hollerith strings are not recognized. Some keywords may be highlighted incorrectly because Fortran90 has no reserved words.

For further information related to Fortran, see ft-fortran-indent and ft-fortran-plugin.

FREEBASIC ft-freebasic-syntax

FreeBASIC files will be highlighted differently for each of the four available dialects, "fb", "qb", "fblite" and "deprecated". See ft-freebasic-plugin for how to select the correct dialect.

Highlighting is further configurable via the following variables.

Variable	Highlight
freebasic_no_comment_fold	disable multiline comment folding
freebasic_operators	non-alpha operators
freebasic_space_errors	trailing white space and spaces before a <Tab>
freebasic_type_suffixes	QuickBASIC style type suffixes

FVWM CONFIGURATION FILES ft-fvwm-syntax

In order for Vim to recognize Fvwm configuration files that do not match the patterns fvwmrc or fvwm2rc , you must put additional patterns appropriate to your system in your myfiletypes.vim file. For these patterns, you must set the variable "b:fvwm_version" to the major version number of Fvwm, and the 'filetype' option to fvwm.

For example, to make Vim identify all files in /etc/X11/fvwm2/ as Fvwm2 configuration files, add the following:

```
:au! BufNewFile,BufRead /etc/X11/fvwm2/*  let b:fvwm_version = 2 |
                                        \ set filetype=fvwm
```

GSP ft-gsp-syntax

The default coloring style for GSP pages is defined by ft-html-syntax, and the coloring for java code (within java tags or inline between backticks) is defined by ft-java-syntax. The following HTML groups defined in ft-html-syntax are redefined to incorporate and highlight inline java code:

```
htmlString
htmlValue
htmlEndTag
htmlTag
htmlTagN
```

Highlighting should look fine most of the places where you'd see inline java code, but in some special cases it may not. To add another HTML group where you will have inline java code where it does not highlight correctly, just copy the line you want from ft-html-syntax and add gspJava to the contains clause.

The backticks for inline java are highlighted according to the htmlError group to make them easier to see.

GDB ft-gdb-syntax

The GDB syntax file provides syntax folding (see :syn-fold) for comments and block statements. This can be enabled with:

```
:set foldmethod=syntax
```

GROFF ft-groff-syntax

The groff syntax file is a wrapper for ft-nroff-syntax, see the notes under that heading for examples of use and configuration. The purpose of this wrapper is to set up groff syntax extensions by setting the filetype from a modeline or in a personal filetype definitions file (see filetype.txt).

HASKELL ft-haskell-syntax

The Haskell syntax files support plain Haskell code as well as literate Haskell code, the latter in both Bird style and TeX style. The Haskell syntax highlighting will also highlight C preprocessor directives.

If you want to highlight delimiter characters (useful if you have a light-coloured background), add to your vimrc:

```
:let hs_highlight_delimiters = 1
```

To treat True and False as keywords as opposed to ordinary identifiers, add:

```
:let hs_highlight_boolean = 1
```

To also treat the names of primitive types as keywords:

```
:let hs_highlight_types = 1
```

And to treat the names of even more relatively common types as keywords:

```
:let hs_highlight_more_types = 1
```

If you want to highlight the names of debugging functions, put in your vimrc:

```
:let hs_highlight_debug = 1
```

The Haskell syntax highlighting also highlights C preprocessor
directives, and flags lines that start with # but are not valid
directives as erroneous. This interferes with Haskell's syntax for
operators, as they may start with #. If you want to highlight those
as operators as opposed to errors, put in your vimrc:

```
:let hs_allow_hash_operator = 1
```

The syntax highlighting for literate Haskell code will try to
automatically guess whether your literate Haskell code contains
TeX markup or not, and correspondingly highlight TeX constructs
or nothing at all. You can override this globally by putting
in your vimrc

```
:let lhs_markup = none
```

for no highlighting at all, or

```
:let lhs_markup = tex
```

to force the highlighting to always try to highlight TeX markup.
For more flexibility, you may also use buffer local versions of
this variable, so e.g.

```
:let b:lhs_markup = tex
```

will force TeX highlighting for a particular buffer. It has to be
set before turning syntax highlighting on for the buffer or
loading a file.

HTML **ft-html-syntax**

The coloring scheme for tags in the HTML file works as follows.

The <> of opening tags are colored differently than the </> of a closing tag.
This is on purpose! For opening tags the 'Function' color is used, while for
closing tags the 'Identifier' color is used (See syntax.vim to check how those
are defined for you)

Known tag names are colored the same way as statements in C. Unknown tag
names are colored with the same color as the <> or </> respectively which
makes it easy to spot errors

Note that the same is true for argument (or attribute) names. Known attribute
names are colored differently than unknown ones.

Some HTML tags are used to change the rendering of text. The following tags
are recognized by the html.vim syntax coloring file and change the way normal
text is shown: <B> <I> <U> <EM> <STRONG> (<EM> is used as an alias for <I>,
while <STRONG> as an alias for <B>), <H1> - <H6>, <HEAD>, <TITLE> and <A>, but
only if used as a link (that is, it must include a href as in
<A href="somefile.html">).

If you want to change how such text is rendered, you must redefine the
following syntax groups:

- htmlBold
- htmlBoldUnderline
- htmlBoldUnderlineItalic
- htmlUnderline
- htmlUnderlineItalic
- htmlItalic
- htmlTitle for titles
- htmlH1 - htmlH6 for headings

To make this redefinition work you must redefine them all with the exception
of the last two (htmlTitle and htmlH[1-6], which are optional) and define the
following variable in your vimrc (this is due to the order in which the files
are read during initialization)

```
:let html_my_rendering=1
```

If you'd like to see an example download mysyntax.vim at
https://web.archive.org/web/20241129015117/https://www.fleiner.com/vim/download.html

You can also disable this rendering by adding the following line to your
vimrc file:

```
:let html_no_rendering=1
```

By default Vim starts parsing 1.5 times "html_minlines" lines before the first
displayed line, and searches a further "html_maxlines" lines back for a
synchronization point. These values can be configured using:

```
:let html_minlines = 500
:let html_maxlines = 750
```

HTML comments are rather special (see an HTML reference document for the
details), and the syntax coloring scheme will highlight all errors.
However, if you prefer to use the wrong style (starts with <!-- and
ends with -->) you can define

```
:let html_wrong_comments=1
```

JavaScript and Visual Basic embedded inside HTML documents are highlighted as
'Special' with statements, comments, strings and so on colored as in standard
programming languages. Note that only JavaScript and Visual Basic are
currently supported, no other scripting language has been added yet.

Embedded and inlined cascading style sheets (CSS) are highlighted too.

There are several html preprocessor languages out there. html.vim has been
written such that it should be trivial to include it. To do so add the
following two lines to the syntax coloring file for that language
(the example comes from the asp.vim file):

```
runtime! syntax/html.vim
syn cluster htmlPreproc add=asp
```

Now you just need to make sure that you add all regions that contain
the preprocessor language to the cluster htmlPreproc.

html-folding

The HTML syntax file provides syntax folding (see :syn-fold) between start
and end tags. This can be turned on by

```
:let g:html_syntax_folding = 1
:set foldmethod=syntax
```

Note: Syntax folding might slow down syntax highlighting significantly,
especially for large files.

HTML/OS (BY AESTIVA) ft-htmlos-syntax

The coloring scheme for HTML/OS works as follows:

Functions and variable names are the same color by default, because VIM
doesn't specify different colors for Functions and Identifiers. To change
this (which is recommended if you want function names to be recognizable in a
different color) you need to add the following line to your vimrc:

```
:hi Function cterm=bold ctermfg=LightGray
```

Of course, the ctermfg can be a different color if you choose.

Another issues that HTML/OS runs into is that there is no special filetype to
signify that it is a file with HTML/OS coding. You can change this by opening
a file and turning on HTML/OS syntax by doing the following:

```
:set syntax=htmlos
```

Lastly, it should be noted that the opening and closing characters to begin a
block of HTML/OS code can either be << or [[and >> or]], respectively.

IA64 **intel-itanium** **ft-ia64-syntax**

Highlighting for the Intel Itanium 64 assembly language. See ft-asm-syntax
for how to recognize this filetype.

To have *.inc files be recognized as IA64, add this to your vimrc file:

```
:let g:filetype_inc = "ia64"
```

ICALENDAR

ft-icalendar-syntax

Highlighting for iCalendar files (RFC 5545), as used by calendaring applications with an ".ics" extension.

Component boundaries (BEGIN: and END:), property names, property parameters and recurrence rule parts are highlighted separately. A content line may be split across several lines by inserting a line break followed by a space or a tab; such folded lines are highlighted as a single logical line.

Folding of components is provided by the filetype plugin and is off by default; see ft-icalendar-plugin.

INFORM

ft-inform-syntax

Inform highlighting includes symbols provided by the Inform Library, as most programs make extensive use of it. If do not wish Library symbols to be highlighted add this to your vim startup:

```
:let inform_highlight_simple=1
```

By default it is assumed that Inform programs are Z-machine targeted, and highlights Z-machine assembly language symbols appropriately. If you intend your program to be targeted to a Glulx/Glk environment you need to add this to your startup sequence:

```
:let inform_highlight_glulx=1
```

This will highlight Glulx opcodes instead, and also adds glk() to the set of highlighted system functions.

The Inform compiler will flag certain obsolete keywords as errors when it encounters them. These keywords are normally highlighted as errors by Vim. To prevent such error highlighting, you must add this to your startup sequence:

```
:let inform_suppress_obsolete=1
```

By default, the language features highlighted conform to Compiler version 6.30 and Library version 6.11. If you are using an older Inform development environment, you may with to add this to your startup sequence:

```
:let inform_highlight_old=1
```

IDL

idl-syntax

IDL (Interface Definition Language) files are used to define RPC calls. In Microsoft land, this is also used for defining COM interfaces and calls.

IDL's structure is simple enough to permit a full grammar based approach to rather than using a few heuristics. The result is large and somewhat repetitive but seems to work.

There are some Microsoft extensions to idl files that are here. Some of them are disabled by defining idl_no_ms_extensions.

The more complex of the extensions are disabled by defining idl_no_extensions.

Variable	Effect
idl_no_ms_extensions	Disable some of the Microsoft specific extensions
idl_no_extensions	Disable complex extensions
idlsyntax_showerror	Show IDL errors (can be rather intrusive, but quite helpful)
idlsyntax_showerror_soft	Use softer colours by default for errors

JAVA

ft-java-syntax

The java.vim syntax highlighting file offers several options.

In Java 1.0.2, it was never possible to have braces inside parens, so this was flagged as an error. Since Java 1.1, this is possible (with anonymous classes); and, therefore, is no longer marked as an error. If you prefer the old way, put the following line into your Vim startup file:

```
:let g:java_mark_braces_in_parens_as_errors = 1
```

All (exported) public types declared in java.lang are always automatically imported and available as simple names. To highlight them, use:

```
:let g:java_highlight_java_lang_ids = 1
```

You can also generate syntax items for other public and protected types and opt in to highlight some of their names; see java-package-info-url.

Headers of indented function declarations can be highlighted (along with parts
of lambda expressions and method reference expressions), but it depends on how
you write Java code. Two formats are recognized:

1) If you write function declarations that are consistently indented by either
a tab, or a space . . . or eight space character(s), you may want to set one
of

```
:let g:java_highlight_functions = "indent"
:let g:java_highlight_functions = "indent1"
:let g:java_highlight_functions = "indent2"
:let g:java_highlight_functions = "indent3"
:let g:java_highlight_functions = "indent4"
:let g:java_highlight_functions = "indent5"
:let g:java_highlight_functions = "indent6"
:let g:java_highlight_functions = "indent7"
:let g:java_highlight_functions = "indent8"
```

Note that in terms of 'shiftwidth', this is the leftmost step of indentation.

2) However, if you follow the Java guidelines about how functions and types
are supposed to be named (with respect to upper- and lowercase) and there is
any amount of indentation, you may want to set

```
:let g:java_highlight_functions = "style"
```

In addition, you can combine any value of "g:java_highlight_functions" with

```
:let g:java_highlight_signature = 1
```

to have the name of a function with its parameter list parens distinctly
highlighted from its type parameters, return type, and formal parameters; and
to have the parameter list parens of a lambda expression with its arrow
distinctly highlighted from its formal parameters or identifiers.

If neither setting does work for you, but you would still want headers of
function declarations to be highlighted, modify the current syntax definitions
or compose new ones.

Higher-order function types can be hard to parse by eye, so uniformly toning
down some of their components may be of value. Provided that such type names
conform to the Java naming guidelines, you may arrange it with

```
:let g:java_highlight_generics = 1
```

In Java 1.1, the functions System.out.println() and System.err.println()
should only be used for debugging. Consider adding the following definition
in your startup file:

```
:let g:java_highlight_debug = 1
```

to have the bulk of those statements colored as
        *Debug   debugging statements,
and to make some of their own items further grouped and linked:
        *Special   as DebugSpecial,
        *String   as DebugString,
        *Boolean   as DebugBoolean,
        *Type   as DebugType,
which are used for special characters appearing in strings, strings proper,
boolean literals, and special instance references (super, this, null),
respectively.

Javadoc is a program that takes special comments out of Java program files and
creates HTML pages. The standard configuration will highlight this HTML code
similarly to HTML files (see ft-html-syntax). You can even add JavaScript
and CSS inside this code (see below). The HTML rendering and the Markdown
rendering diverge as follows:
 1. The first sentence (all characters up to the first period ., which is

       followed by a whitespace character or a line terminator, or up to the
       first block tag, e.g. @param, @return) is colored as
          *SpecialComment   special comments.
 2. The text is colored as

          *Comment   comments.
 3. HTML comments are colored as

          *Special   special symbols.
 4. The standard Javadoc tags (@code, @see, etc.) are colored as

          *Special   special symbols
       and some of their arguments are colored as
          *Function   function names.

To turn this feature off for both HTML and Markdown, add the following line to your startup file:

```
:let g:java_ignore_javadoc = 1
```

Alternatively, only suppress HTML comments or Markdown comments:

```
:let g:java_ignore_html = 1
:let g:java_ignore_markdown = 1
```

See ft-java-plugin for additional support available for Markdown comments.

If you use the special Javadoc comment highlighting described above, you can also turn on special highlighting for JavaScript, Visual Basic scripts, and embedded CSS (stylesheets). This only makes sense if any of these languages actually appear in Javadoc comments. The variables to use are

```
:let g:java_javascript = 1
:let g:java_css = 1
:let g:java_vb = 1
```

Note that these three variables are maintained in the HTML syntax file.

Numbers and strings can be recognized in non-Javadoc comments with

```
:let g:java_comment_strings = 1
```

When 'foldmethod' is set to "syntax", multi-line blocks of code ("b"), plain comments ("c"), Javadoc comments ("d"), and adjacent "import" declarations ("i") will be folded by default. Syntax items of any supported kind will remain NOT foldable when its abbreviated name is delisted with

```
:let g:java_ignore_folding = "bcdi"
```

No text is usually written in the first line of a multi-line comment, making folded contents of Javadoc comments less informative with the default 'foldtext' value; you may opt for showing the contents of a second line for any comments written in this way, and showing the contents of a first line otherwise, with

```
:let g:java_foldtext_show_first_or_second_line = 1
```

HTML tags in Javadoc comments can additionally be folded by following the instructions listed under html-folding and giving explicit consent with

```
:let g:java_consent_to_html_syntax_folding = 1
```

Do not default to this kind of folding unless ALL start tags and optional end tags are balanced in Javadoc comments; otherwise, put up with creating runaway folds that break syntax highlighting.

Trailing whitespace characters or a run of space characters before a tab character can be marked as an error with

```
:let g:java_space_errors = 1
```

but either kind of an error can be suppressed by also defining one of

```
:let g:java_no_trail_space_error = 1
:let g:java_no_tab_space_error = 1
```

In order to highlight nested parens with different colors, define colors for javaParen, javaParen1, and javaParen2. For example,

```
:hi link javaParen Comment
```

or

```
:hi javaParen ctermfg=blue guifg=#0000ff
```

Certain modifiers are incompatible with each other, e.g. abstract and final:

```
:syn list javaConceptKind
```

and can be differently highlighted as a group than other modifiers with

```
:hi link javaConceptKind NonText
```

All instances of variable-width lookbehind assertions (/\@<! and /\@<=),
resorted to in syntax item definitions, are confined to arbitrary byte counts.
Another arbitrary value can be selected for a related group of definitions.
For example:

```
:let g:java_lookbehind_byte_counts = {'javaMarkdownCommentTitle': 240}
```

Where each key name of this dictionary is the name of a syntax item. The use
of these assertions in syntax items may vary among revisions, so no definitive
set of supported key names is committed to.

If you notice highlighting errors while scrolling backwards, which are fixed
when redrawing with CTRL-L, try setting the "g:java_minlines" variable to
a larger number:

```
:let g:java_minlines = 50
```

This will make the syntax synchronization start 50 lines before the first
displayed line. The default value is 10. The disadvantage of using a larger
number is that redrawing can become slow.

Significant changes to the Java platform are gradually introduced in the form
of JDK Enhancement Proposals (JEPs) that can be implemented for a release and
offered as its preview features. It may take several JEPs and a few release
cycles for such a feature to become either integrated into the platform or
withdrawn from this effort. To cater for early adopters, there is optional
support in Vim for syntax related preview features that are implemented. You
can request it by specifying a list of preview feature numbers as follows:

```
:let g:java_syntax_previews = [532]
```

The supported JEP numbers are to be drawn from this table:
        430: String Templates [JDK 21]
        532: Primitive types in Patterns, instanceof, and switch

Note that as soon as the particular preview feature will have been integrated
into the Java platform, its entry will be removed from the table and related
optionality will be discontinued.

java-package-info-url

https://github.com/zzzyxwvut/java-vim/blob/master/tools/javaid/src/javaid/package-info.java

JSON ft-json-syntax g:vim_json_conceal

g:vim_json_warnings

The json syntax file provides syntax highlighting with conceal support by
default. To disable concealment:

```
let g:vim_json_conceal = 0
```

To disable syntax highlighting of errors:

```
let g:vim_json_warnings = 0
```

JQ jq_quote_highlight ft-jq-syntax

To disable numbers having their own color add the following to your vimrc:

```
hi link jqNumber Normal
```

If you want quotes to have different highlighting than strings

```
let g:jq_quote_highlight = 1
```

KCONFIG ft-kconfig-syntax

Kconfig syntax highlighting language. For syntax syncing, you can configure
the following variable (default: 50):

```
let kconfig_minlines = 50
```

To configure a bit more (heavier) highlighting, set the following variable:

```
let kconfig_syntax_heavy = 1
```

LACE ft-lace-syntax

Lace (Language for Assembly of Classes in Eiffel) is case insensitive, but the
style guide lines are not. If you prefer case insensitive highlighting, just

define the vim variable 'lace_case_insensitive' in your startup file:

```
:let lace_case_insensitive=1
```

LF (LFRC)

ft-lf-syntax g:lf_shell_syntax

b:lf_shell_syntax

For the lf file manager configuration files (lfrc) the shell commands syntax
highlighting can be changed globally and per buffer by setting a different
'include' command search pattern using these variables:

```
let g:lf_shell_syntax = "syntax/dosbatch.vim"
let b:lf_shell_syntax = "syntax/zsh.vim"
```

These variables are unset by default.

The default 'include' command search pattern is 'syntax/sh.vim'.

LEX

ft-lex-syntax

Lex uses brute-force synchronizing as the "^%%$" section delimiter
gives no clue as to what section follows. Consequently, the value for

```
:syn sync minlines=300
```

may be changed by the user if they are experiencing synchronization
difficulties (such as may happen with large lex files).

LIFELINES

ft-lifelines-syntax

To highlight deprecated functions as errors, add in your vimrc:

```
:let g:lifelines_deprecated = 1
```

LISP

ft-lisp-syntax

The lisp syntax highlighting provides two options:

```
g:lisp_instring : If it exists, then "(...)" strings are highlighted
                  as if the contents of the string were lisp.
                  Useful for AutoLisp.
g:lisp_rainbow  : If it exists and is nonzero, then differing levels
                  of parenthesization will receive different
                  highlighting.
```

The g:lisp_rainbow option provides 10 levels of individual colorization for
the parentheses and backquoted parentheses. Because of the quantity of
colorization levels, unlike non-rainbow highlighting, the rainbow mode
specifies its highlighting using ctermfg and guifg, thereby bypassing the
usual color scheme control using standard highlighting groups. The actual
highlighting used depends on the dark/bright setting (see 'bg').

LITE

ft-lite-syntax

There are two options for the lite syntax highlighting.

If you like SQL syntax highlighting inside Strings, use this:

```
:let lite_sql_query = 1
```

For syncing, minlines defaults to 100. If you prefer another value, you can
set "lite_minlines" to the value you desire. Example:

```
:let lite_minlines = 200
```

LOG

ft-log-syntax

Vim comes with a simplistic generic log syntax highlighter. Because the log
file format is so generic, highlighting is not enabled by default, since it
can be distracting. If you want to enable this, simply set the "log" filetype
manually:

```
:set ft=log
```

To enable this automatically for "*.log" files, add the following to your
personal filetype.vim file (usually located in ~/.config/nvim/filetype.vim
on Unix, see also new-filetype):

```
augroup filetypedetect
  au! BufNewFile,BufRead *.log  setfiletype log
augroup END
```

LPC

ft-lpc-syntax

LPC stands for a simple, memory-efficient language: Lars Pensjö C. The file name of LPC is usually *.c. Recognizing these files as LPC would bother users writing only C programs. If you want to use LPC syntax in Vim, you should set a variable in your vimrc file:

```
:let lpc_syntax_for_c = 1
```

If it doesn't work properly for some particular C or LPC files, use a modeline. For a LPC file:

```
// vim:set ft=lpc:
```

For a C file that is recognized as LPC:

```
// vim:set ft=c:
```

If you don't want to set the variable, use the modeline in EVERY LPC file.

There are several implementations for LPC, we intend to support most widely used ones. Here the default LPC syntax is for MudOS series, for MudOS v22 and before, you should turn off the sensible modifiers, and this will also assert the new efuns after v22 to be invalid, don't set this variable when you are using the latest version of MudOS:

```
:let lpc_pre_v22 = 1
```

For LpMud 3.2 series of LPC:

```
:let lpc_compat_32 = 1
```

For LPC4 series of LPC:

```
:let lpc_use_lpc4_syntax = 1
```

For uLPC series of LPC:
uLPC has been developed to Pike, so you should use Pike syntax instead, and the name of your source file should be *.pike

LUA

ft-lua-syntax

The Lua syntax file can be used for versions 4.0, 5.0+. You can select one of these versions using the global variables g:lua_version and g:lua_subversion.

MAIL

ft-mail.vim

Vim highlights all the standard elements of an email (headers, signatures, quoted text and URLs / email addresses). In keeping with standard conventions, signatures begin in a line containing only "--" followed optionally by whitespaces and end with a newline.

Vim treats lines beginning with "]", "}", "|", ">" or a word followed by ">" as quoted text. However Vim highlights headers and signatures in quoted text only if the text is quoted with ">" (optionally followed by one space).

By default mail.vim synchronises syntax to 100 lines before the first displayed line. If you have a slow machine, and generally deal with emails with short headers, you can change this to a smaller value:

```
:let mail_minlines = 30
```

MAKE

ft-make-syntax

In makefiles, commands are usually highlighted to make it easy for you to spot errors. However, this may be too much coloring for you. You can turn this feature off by using:

```
:let make_no_commands = 1
```

Comments are also highlighted by default. You can turn this off by using:

```
:let make_no_comments = 1
```

There are various Make implementations, which add extensions other than the POSIX specification and thus are mutually incompatible. If the filename is BSDmakefile or GNUmakefile, the corresponding implementation is automatically determined; otherwise vim tries to detect it by the file contents. If you see any wrong highlights because of this, you can enforce a flavor by setting one of the following:

```
:let g:make_flavor = 'bsd'  " or
:let g:make_flavor = 'gnu'  " or
:let g:make_flavor = 'microsoft'
```

MAPLE ft-maple-syntax

Maple V, by Waterloo Maple Inc, supports symbolic algebra. The language supports many packages of functions which are selectively loaded by the user. The standard set of packages' functions as supplied in Maple V release 4 may be highlighted at the user's discretion. Users may place in their vimrc file:

```
:let mvpkg_all= 1
```

to get all package functions highlighted, or users may select any subset by choosing a variable/package from the table below and setting that variable to 1, also in their vimrc file (prior to sourcing $VIMRUNTIME/syntax/syntax.vim).

Table of Maple V Package Function Selectors

```
mv_DEtools       mv_genfunc     mv_networks     mv_process
mv_Galois        mv_geometry    mv_numapprox    mv_simplex
mv_GaussInt      mv_grobner     mv_numtheory    mv_stats
mv_LREtools      mv_group       mv_orthopoly    mv_student
mv_combinat      mv_inttrans    mv_padic        mv_sumtools
mv_combstruct mv_liesymm        mv_plots        mv_tensor
mv_difforms      mv_linalg      mv_plottools    mv_totorder
mv_finance       mv_logic       mv_powseries
```

MARKDOWN ft-markdown-syntax g:markdown_minlines

g:markdown_fenced_languages g:markdown_syntax_conceal

If you have long regions there may be incorrect highlighting. At the cost of slowing down displaying, you can have the engine look further back to sync on the start of a region, for example 500 lines (default is 50):

```
:let g:markdown_minlines = 500
```

If you want to enable fenced code block syntax highlighting in your Markdown documents, set the following variable:

```
:let g:markdown_fenced_languages = ['html', 'python', 'bash=sh']
```

To disable Markdown syntax concealing, add the following to your vimrc:

```
:let g:markdown_syntax_conceal = 0
```

For extended Markdown support with enhanced features such as citations, footnotes, mathematical formulas, academic writing elements and embedded code block highlighting, consider using the pandoc syntax plugin. Set g:filetype_md to "pandoc" and see ft-pandoc-syntax for configuration details.

MATHEMATICA ft-mma-syntax ft-mathematica-syntax

Empty *.m files will automatically be presumed to be Matlab files unless you have the following in your vimrc:

```
let filetype_m = "mma"
```

MBSYNC ft-mbsync-syntax

The mbsync application uses a configuration file to setup mailboxes names, user and password. All files ending with .mbsyncrc or with the name isyncrc will be recognized as mbsync configuration files.

MEDIAWIKI ft-mediawiki-syntax

By default, syntax highlighting includes basic HTML tags like style and headers ft-html-syntax. For strict Mediawiki syntax highlighting:

```
let g:html_no_rendering = 1
```

If HTML highlighting is desired, terminal-based text formatting such as bold and italic is possible by:

```
let g:html_style_rendering = 1
```

MODULA2

ft-modula2-syntax

Vim will recognise comments with dialect tags to automatically select a given dialect.

The syntax for a dialect tag comment is:

```
taggedComment :=
  '(*!' dialectTag '*)'
  ;

dialectTag :=
  m2pim | m2iso | m2r10
  ;

reserved words
  m2pim = 'm2pim', m2iso = 'm2iso', m2r10 = 'm2r10'
```

A dialect tag comment is recognised by Vim if it occurs within the first 200 lines of the source file. Only the very first such comment is recognised, any additional dialect tag comments are ignored.

Example:

```
DEFINITION MODULE FooLib; (*!m2pim*)
...
```

Variable g:modula2_default_dialect sets the default Modula-2 dialect when the dialect cannot be determined from the contents of the Modula-2 file: if defined and set to 'm2pim', the default dialect is PIM.

Example:

```
let g:modula2_default_dialect = 'm2pim'
```

Highlighting is further configurable for each dialect via the following variables.

Variable	Highlight
modula2_iso_allow_lowline	allow low line in identifiers
modula2_iso_disallow_octals	disallow octal integer literals
modula2_iso_disallow_synonyms	disallow "@", "&" and "~" synonyms

```
modula2_pim_allow_lowline       allow low line in identifiers
modula2_pim_disallow_octals     disallow octal integer literals
modula2_pim_disallow_synonyms   disallow "&" and "~" synonyms

modula2_r10_allow_lowline       allow low line in identifiers
```

MOO

ft-moo-syntax

If you use C-style comments inside expressions and find it mangles your highlighting, you may want to use extended (slow!) matches for C-style comments:

```
:let moo_extended_cstyle_comments = 1
```

To disable highlighting of pronoun substitution patterns inside strings:

```
:let moo_no_pronoun_sub = 1
```

To disable highlighting of the regular expression operator "%|", and matching "%(" and "%)" inside strings:

```
:let moo_no_regexp = 1
```

Unmatched double quotes can be recognized and highlighted as errors:

```
:let moo_unmatched_quotes = 1
```

To highlight builtin properties (.name, .location, .programmer etc.):

```
:let moo_builtin_properties = 1
```

Unknown builtin functions can be recognized and highlighted as errors. If you use this option, add your own extensions to the mooKnownBuiltinFunction group. To enable this option:

```
:let moo_unknown_builtin_functions = 1
```

An example of adding sprintf() to the list of known builtin functions:

```
:syn keyword mooKnownBuiltinFunction sprintf contained
```

MSQL ft-msql-syntax

There are two options for the msql syntax highlighting.

If you like SQL syntax highlighting inside Strings, use this:

```
:let msql_sql_query = 1
```

For syncing, minlines defaults to 100. If you prefer another value, you can set "msql_minlines" to the value you desire. Example:

```
:let msql_minlines = 200
```

NEOMUTT ft-neomuttrc-syntax

ft-neomuttlog-syntax

To disable the default NeoMutt log colors:

```
:let g:neolog_disable_default_colors = 1
```

N1QL ft-n1ql-syntax

N1QL is a SQL-like declarative language for manipulating JSON documents in Couchbase Server databases.

Vim syntax highlights N1QL statements, keywords, operators, types, comments, and special values. Vim ignores syntactical elements specific to SQL or its many dialects, like COLUMN or CHAR, that don't exist in N1QL.

NCF ft-ncf-syntax

There is one option for NCF syntax highlighting.

If you want to have unrecognized (by ncf.vim) statements highlighted as errors, use this:

```
:let ncf_highlight_unknowns = 1
```

If you don't want to highlight these errors, leave it unset.

NROFF ft-nroff-syntax

The nroff syntax file works with AT&T n/troff as-is. To support GNU troff (groff), which Linux and BSD distributions use as their default typesetting package, arrange for files to be recognized as groff input (see ft-groff-syntax) or add the following option to your start-up files:

```
:let nroff_is_groff = 1
```

GNU troff differs from older AT&T n/troff programs (that you may still find in Solaris or Plan 9) by extending the "*roff" language syntax. For example, in AT&T troff, you access the count of years since 1900 with the escape sequence \n(yr. In groff you can do the same, which it recognizes for compatibility, or use groff's extended syntax, \n[yr]. AT&T troff documented the yr register as storing the "last two digits of current year", but had a Y2K problem; in groff, you can access the Gregorian year correctly: \n[year]. In groff, font, register, macro, string, and request names can exceed two characters; for example, with groff's mm package, the control lines ".VERBON" and ".VERBOFF" call macros of those names to bracket displays of "verbatim" content.

In order to obtain the best formatted output g/troff can give you, you should follow a few simple rules about spacing and punctuation.

1. Break the line (put a carriage return) at the end of every sentence. Don't permit trailing spaces before the newline.
2. If a line ends with a period, question mark, or exclamation point that does not end a sentence, follow it with the dummy character escape sequence \&.
3. If you're using a macro package, employ its paragraphing macros to achieve indentation of paragraphs and spacing between them.
4. Use the empty request, a '.' on a line by itself, freely to visually separate material for ease of document maintenance.

The reason for these tips is that g/n/troff attempts to detect the ends of sentences, and can use that information to apply inter-sentence space. Using them also minimizes the size of diffs where lines change due only to refilling in the text editor. Macro packages typically employ inter-paragraph spacing amounts other than one vee (which is the result of a blank input line), and typically store that spacing amount, and that of paragraph indentation, in user-configurable registers so that pages lay out consistently.

Unlike TeX, troff fills text line-by-line, not paragraph-by-paragraph. If you desire consistent spacing between words and sentences in formatted output, you must maintain consistent spacing in the input text. To mark both trailing spaces and two or more spaces after a punctuation as an error, use:

```
:let nroff_space_errors = 1
```

Another technique to detect extra spacing and other errors that will interfere with the correct typesetting of your file, is to define an eye-catching highlighting definition for the syntax groups "nroffDefinition" and "nroffDefSpecial" in your configuration files. For example:

```
hi def nroffDefinition cterm=italic gui=reverse
hi def nroffDefSpecial cterm=italic,bold gui=reverse,bold
```

If you want to navigate preprocessor entries in your source file as easily as with section markers, you can activate the following option in your vimrc file:

```
let b:preprocs_as_sections = 1
```

Further, the syntax file adds an extra paragraph marker for the XP paragraphing macro in the ms package, a Berkeley and GNU extension.

Finally, there is a ft-groff-syntax file that can be used to enable groff syntax highlighting either on a per-file basis or globally by default.

OCAML **ft-ocaml-syntax**

The OCaml syntax file handles files having the following prefixes: .ml, .mli, .mll and .mly. By setting the following variable

```
:let ocaml_revised = 1
```

you can switch from standard OCaml-syntax to revised syntax as supported by the camlp4 preprocessor. Setting the variable

```
:let ocaml_noend_error = 1
```

prevents highlighting of "end" as error, which is useful when sources contain very long structures that Vim does not synchronize anymore.

PANDOC **ft-pandoc-syntax**

By default, markdown files will be detected as filetype "markdown". Alternatively, you may want them to be detected as filetype "pandoc" instead. To do so, set the g:filetype_md var:

```
:let g:filetype_md = 'pandoc'
```

The pandoc syntax plugin uses conceal for pretty highlighting. Default is 1

```
:let g:pandoc#syntax#conceal#use = 1
```

To specify elements that should not be concealed, set the following variable:

```
:let g:pandoc#syntax#conceal#blacklist = []
```

This is a list of the rules which can be used here:

- titleblock
- image
- block
- subscript
- superscript

- strikeout
- atx
- codeblock_start
- codeblock_delim
- footnote
- definition
- list
- newline
- dashes
- ellipses
- quotes
- inlinecode
- inlinemath

You can customize the way concealing works. For example, if you prefer to
mark footnotes with the * symbol:

```
:let g:pandoc#syntax#conceal#cchar_overrides = {"footnote" : "*"}
```

To conceal the urls in links, use:

```
:let g:pandoc#syntax#conceal#urls = 1
```

Prevent highlighting specific codeblock types so that they remain Normal.
Codeblock types include "definition" for codeblocks inside definition blocks
and "delimited" for delimited codeblocks. Default = []

```
:let g:pandoc#syntax#codeblocks#ignore = ['definition']
```

Use embedded highlighting for delimited codeblocks where a language is
specified. Default = 1

```
:let g:pandoc#syntax#codeblocks#embeds#use = 1
```

For specify what languages and using what syntax files to highlight embeds.
This is a list of language names. When the language pandoc and vim use don't
match, you can use the "PANDOC=VIM" syntax. For example:

```
:let g:pandoc#syntax#codeblocks#embeds#langs = ["ruby", "bash=sh"]
```

To use italics and strong in emphases. Default = 1

```
:let g:pandoc#syntax#style#emphases = 1
```

"0" will add "block" to g:pandoc#syntax#conceal#blacklist, because otherwise
you couldn't tell where the styles are applied.

To add underline subscript, superscript and strikeout text styles. Default = 1

```
:let g:pandoc#syntax#style#underline_special = 1
```

Detect and highlight definition lists. Disabling this can improve
performance. Default = 1 (i.e., enabled by default)

```
:let g:pandoc#syntax#style#use_definition_lists = 1
```

The pandoc syntax script also comes with the following commands:

```
:PandocHighlight LANG
```

Enables embedded highlighting for language LANG in codeblocks. Uses the
syntax for items in g:pandoc#syntax#codeblocks#embeds#langs.

```
:PandocUnhighlight LANG
```

Disables embedded highlighting for language LANG in codeblocks.

PAPP **ft-papp-syntax**

The PApp syntax file handles .papp files and, to a lesser extent, .pxml
and .pxsl files which are all a mixture of perl/xml/html/other using xml
as the top-level file format. By default everything inside phtml or pxml
sections is treated as a string with embedded preprocessor commands. If
you set the variable:

```
:let papp_include_html=1
```

in your startup file it will try to syntax-highlight html code inside phtml
sections, but this is relatively slow and much too colourful to be able to
edit sensibly. ;)

The newest version of the papp.vim syntax file can usually be found at
http://papp.plan9.de.

PASCAL **ft-pascal-syntax**

Files matching "*.p" could be Progress or Pascal and those matching "*.pp"
could be Puppet or Pascal. If the automatic detection doesn't work for you,
or you only edit Pascal files, use this in your startup vimrc:

```
:let filetype_p  = "pascal"
:let filetype_pp = "pascal"
```

The Pascal syntax file has been extended to take into account some extensions
provided by Turbo Pascal, Free Pascal Compiler and GNU Pascal Compiler.
Delphi keywords are also supported. By default, Turbo Pascal 7.0 features are
enabled. If you prefer to stick with the standard Pascal keywords, add the
following line to your startup file:

```
:let pascal_traditional=1
```

To switch on Delphi specific constructions (such as one-line comments,
keywords, etc):

```
:let pascal_delphi=1
```

The option pascal_symbol_operator controls whether symbol operators such as +,
*, .., etc. are displayed using the Operator color or not. To colorize symbol
operators, add the following line to your startup file:

```
:let pascal_symbol_operator=1
```

Some functions are highlighted by default. To switch it off:

```
:let pascal_no_functions=1
```

Furthermore, there are specific variables for some compilers. Besides
pascal_delphi, there are pascal_gpc and pascal_fpc. Default extensions try to
match Turbo Pascal.

```
:let pascal_gpc=1
```

or

```
:let pascal_fpc=1
```

To ensure that strings are defined on a single line, you can define the
pascal_one_line_string variable.

```
:let pascal_one_line_string=1
```

If you dislike <Tab> chars, you can set the pascal_no_tabs variable. Tabs
will be highlighted as Error.

```
:let pascal_no_tabs=1
```

PERL ft-perl-syntax

There are a number of possible options to the perl syntax highlighting.

Inline POD highlighting is now turned on by default. If you don't wish to have the added complexity of highlighting POD embedded within Perl files, you may set the 'perl_include_pod' option to 0:

```
:let perl_include_pod = 0
```

To reduce the complexity of parsing (and increase performance) you can switch off two elements in the parsing of variable names and contents.

To handle package references in variable and function names not differently from the rest of the name (like 'PkgName::' in '$PkgName::VarName'):

```
:let perl_no_scope_in_variables = 1
```

(In Vim 6.x it was the other way around: "perl_want_scope_in_variables" enabled it.)

If you do not want complex things like @{${"foo"}} to be parsed:

```
:let perl_no_extended_vars = 1
```

(In Vim 6.x it was the other way around: "perl_extended_vars" enabled it.)

The coloring strings can be changed. By default strings and qq friends will be highlighted like the first line. If you set the variable perl_string_as_statement, it will be highlighted as in the second line.

```
   "hello world!"; qq|hello world|;
   ^^^^^^^^^^^^^^NN^^^^^^^^^^^^^^^N       (unlet perl_string_as_statement)
   S^^^^^^^^^^^^SNNSSS^^^^^^^^^^^SN       (let perl_string_as_statement)
```

(^ = perlString, S = perlStatement, N = None at all)

The syncing has 3 options. The first two switch off some triggering of synchronization and should only be needed in case it fails to work properly. If while scrolling all of a sudden the whole screen changes color completely then you should try and switch off one of those. Let the developer know if you can figure out the line that causes the mistake.

One triggers on "^\s*sub\s*" and the other on "^[$@%]" more or less.

```
:let perl_no_sync_on_sub
:let perl_no_sync_on_global_var
```

Below you can set the maximum distance VIM should look for starting points for its attempts in syntax highlighting.

```
:let perl_sync_dist = 100
```

If you want to use folding with perl, set perl_fold:

```
:let perl_fold = 1
```

If you want to fold blocks in if statements, etc. as well set the following:

```
:let perl_fold_blocks = 1
```

Subroutines are folded by default if 'perl_fold' is set. If you do not want this, you can set 'perl_nofold_subs':

```
:let perl_nofold_subs = 1
```

Anonymous subroutines are not folded by default; you may enable their folding via 'perl_fold_anonymous_subs':

```
:let perl_fold_anonymous_subs = 1
```

Packages are also folded by default if 'perl_fold' is set. To disable this behavior, set 'perl_nofold_packages':

```
:let perl_nofold_packages = 1
```

PHP3 and PHP4 ft-php-syntax ft-php3-syntax

[**Note:** Previously this was called "php3", but since it now also supports php4 it has been renamed to "php"]

There are the following options for the php syntax highlighting.

If you like SQL syntax highlighting inside Strings:

```
let php_sql_query = 1
```

For highlighting the Baselib methods:

```
let php_baselib = 1
```

Enable HTML syntax highlighting inside strings:

```
let php_htmlInStrings = 1
```

Using the old colorstyle:

```
let php_oldStyle = 1
```

Enable highlighting ASP-style short tags:

```
let php_asp_tags = 1
```

Disable short tags:

```
let php_noShortTags = 1
```

For highlighting parent error] or):

```
let php_parent_error_close = 1
```

For skipping a php end tag, if there exists an open (or [without a closing
one:

```
let php_parent_error_open = 1
```

Enable folding for classes and functions:

```
let php_folding = 1
```

Selecting syncing method:

```
let php_sync_method = x
```

x = -1 to sync by search (default),
x > 0 to sync at least x lines backwards,
x = 0 to sync from start.

PLAINTEX ft-plaintex-syntax

TeX is a typesetting language, and plaintex is the file type for the "plain"
variant of TeX. If you never want your *.tex files recognized as plain TeX,
see ft-tex-plugin.

This syntax file has the option

```
let g:plaintex_delimiters = 1
```

if you want to highlight brackets "[]" and braces "{}".

PPWIZARD ft-ppwiz-syntax

PPWizard is a preprocessor for HTML and OS/2 INF files

This syntax file has the options:

- ppwiz_highlight_defs : Determines highlighting mode for PPWizard's

  definitions. Possible values are

  ppwiz_highlight_defs = 1 : PPWizard #define statements retain the
  colors of their contents (e.g. PPWizard macros and variables).

  ppwiz_highlight_defs = 2 : Preprocessor #define and #evaluate
  statements are shown in a single color with the exception of line
  continuation symbols.

  The default setting for ppwiz_highlight_defs is 1.

- ppwiz_with_html : If the value is 1 (the default), highlight literal

  HTML code; if 0, treat HTML code like ordinary text.

PHTML ft-phtml-syntax

There are two options for the phtml syntax highlighting.

If you like SQL syntax highlighting inside Strings, use this:

```
:let phtml_sql_query = 1
```

For syncing, minlines defaults to 100. If you prefer another value, you can set "phtml_minlines" to the value you desire. Example:

```
:let phtml_minlines = 200
```

POSTSCRIPT ft-postscr-syntax

There are several options when it comes to highlighting PostScript.

First which version of the PostScript language to highlight. There are currently three defined language versions, or levels. Level 1 is the original and base version, and includes all extensions prior to the release of level 2. Level 2 is the most common version around, and includes its own set of extensions prior to the release of level 3. Level 3 is currently the highest level supported. You select which level of the PostScript language you want highlighted by defining the postscr_level variable as follows:

```
:let postscr_level=2
```

If this variable is not defined it defaults to 2 (level 2) since this is the most prevalent version currently.

Note: Not all PS interpreters will support all language features for a particular language level. In particular the %!PS-Adobe-3.0 at the start of PS files does NOT mean the PostScript present is level 3 PostScript!

If you are working with Display PostScript, you can include highlighting of Display PS language features by defining the postscr_display variable as follows:

```
:let postscr_display=1
```

If you are working with Ghostscript, you can include highlighting of Ghostscript specific language features by defining the variable postscr_ghostscript as follows:

```
:let postscr_ghostscript=1
```

PostScript is a large language, with many predefined elements. While it useful to have all these elements highlighted, on slower machines this can cause Vim to slow down. In an attempt to be machine friendly font names and character encodings are not highlighted by default. Unless you are working explicitly with either of these this should be ok. If you want them to be highlighted you should set one or both of the following variables:

```
:let postscr_fonts=1
:let postscr_encodings=1
```

There is a stylistic option to the highlighting of and, or, and not. In PostScript the function of these operators depends on the types of their operands - if the operands are booleans then they are the logical operators, if they are integers then they are binary operators. As binary and logical operators can be highlighted differently they have to be highlighted one way or the other. By default they are treated as logical operators. They can be highlighted as binary operators by defining the variable postscr_andornot_binary as follows:

```
:let postscr_andornot_binary=1
```

ft-printcap-syntax

PRINTCAP + TERMCAP ft-ptcap-syntax ft-termcap-syntax

This syntax file applies to the printcap and termcap databases.

In order for Vim to recognize printcap/termcap files that do not match the patterns "*printcap*", or "*termcap*", you must put additional patterns appropriate to your system in your myfiletypefile file. For these patterns, you must set the variable "b:ptcap_type" to either "print" or "term", and then the 'filetype' option to ptcap.

For example, to make Vim identify all files in /etc/termcaps/ as termcap files, add the following:

```
:au BufNewFile,BufRead /etc/termcaps/* let b:ptcap_type = "term" |
                                    \ set filetype=ptcap
```

If you notice highlighting errors while scrolling backwards, which

are fixed when redrawing with CTRL-L, try setting the "ptcap_minlines"
internal variable to a larger number:

```
:let ptcap_minlines = 50
```

(The default is 20 lines.)

PROGRESS ft-progress-syntax

Files matching "*.w" could be Progress or cweb. If the automatic detection
doesn't work for you, or you don't edit cweb at all, use this in your
startup vimrc:

```
:let filetype_w = "progress"
```

The same happens for "*.i", which could be assembly, and "*.p", which could be
Pascal. Use this if you don't use assembly and Pascal:

```
:let filetype_i = "progress"
:let filetype_p = "progress"
```

PYTHON ft-python-syntax

There are seven options to control Python syntax highlighting.

For highlighted numbers:

```
:let python_no_number_highlight = 1
```

For highlighted builtin functions:

```
:let python_no_builtin_highlight = 1
```

For highlighted standard exceptions:

```
:let python_no_exception_highlight = 1
```

For highlighted doctests and code inside:

```
:let python_no_doctest_highlight = 1
```

or

```
:let python_no_doctest_code_highlight = 1
```

The first option implies the second one.

For highlighted trailing whitespace and mix of spaces and tabs:

```
:let python_space_error_highlight = 1
```

For highlighted built-in constants distinguished from other keywords:

```
:let python_constant_highlight = 1
```

If you want all possible Python highlighting:

```
:let python_highlight_all = 1
```

This has the same effect as setting python_space_error_highlight,
python_constant_highlight and unsetting all the other ones.

If you use Python 2 or straddling code (Python 2 and 3 compatible),
you can enforce the use of an older syntax file with support for
Python 2 and up to Python 3.5.

```
:let python_use_python2_syntax = 1
```

This option will exclude all modern Python 3.6 or higher features.

Note: Only existence of these options matters, not their value.
You can replace 1 above with anything.

QUAKE ft-quake-syntax

The Quake syntax definition should work for most FPS (First Person Shooter)
based on one of the Quake engines. However, the command names vary a bit
between the three games (Quake, Quake 2, and Quake 3 Arena) so the syntax
definition checks for the existence of three global variables to allow users

to specify what commands are legal in their files. The three variables can
be set for the following effects:

set to highlight commands only available in Quake:

```
:let quake_is_quake1 = 1
```

set to highlight commands only available in Quake 2:

```
:let quake_is_quake2 = 1
```

set to highlight commands only available in Quake 3 Arena:

```
:let quake_is_quake3 = 1
```

Any combination of these three variables is legal, but might highlight more
commands than are actually available to you by the game.

R ft-r-syntax

The parsing of R code for syntax highlight starts 40 lines backwards, but you
can set a different value in your vimrc. Example:

```
let r_syntax_minlines = 60
```

You can also turn off syntax highlighting of ROxygen:

```
let r_syntax_hl_roxygen = 0
```

enable folding of code delimited by parentheses, square brackets and curly
braces:

```
let r_syntax_folding = 1
```

and highlight as functions all keywords followed by an opening parenthesis:

```
let r_syntax_fun_pattern = 1
```

R MARKDOWN ft-rmd-syntax

To disable syntax highlight of YAML header, add to your vimrc:

```
let rmd_syn_hl_yaml = 0
```

To disable syntax highlighting of citation keys:

```
let rmd_syn_hl_citations = 0
```

To highlight R code in knitr chunk headers:

```
let rmd_syn_hl_chunk = 1
```

By default, chunks of R code will be highlighted following the rules of R
language. Moreover, whenever the buffer is saved, Vim scans the buffer and
highlights other languages if they are present in new chunks. LaTeX code also
is automatically recognized and highlighted when the buffer is saved. This
behavior can be controlled with the variables rmd_dynamic_fenced_languages,
and rmd_include_latex whose valid values are:

```
let rmd_dynamic_fenced_languages = 0 " No autodetection of languages
let rmd_dynamic_fenced_languages = 1 " Autodetection of languages
let rmd_include_latex = 0 " Don't highlight LaTeX code
let rmd_include_latex = 1 " Autodetect LaTeX code
let rmd_include_latex = 2 " Always include LaTeX highlighting
```

If the value of rmd_dynamic_fenced_languages is 0, you still can set the
list of languages whose chunks of code should be properly highlighted, as in
the example:

```
let rmd_fenced_languages = ['r', 'python']
```

R RESTRUCTURED TEXT ft-rrst-syntax

To highlight R code in knitr chunk headers, add to your vimrc:

```
let rrst_syn_hl_chunk = 1
```

RASI ft-rasi-syntax

Rasi stands for Rofi Advanced Style Information. It is used by the program rofi to style the rendering of the search window. The language is heavily inspired by CSS stylesheet. Files with the following extensions are recognized as rasi files: .rasi.

READLINE ft-readline-syntax

The readline library is primarily used by the BASH shell, which adds quite a few commands and options to the ones already available. To highlight these items as well you can add the following to your vimrc or just type it in the command line before loading a file with the readline syntax:

```
let readline_has_bash = 1
```

This will add highlighting for the commands that BASH (version 2.05a and later, and part earlier) adds.

REGO ft-rego-syntax

Rego is a query language developed by Styra. It is mostly used as a policy language for kubernetes, but can be applied to almost anything. Files with the following extensions are recognized as rego files: .rego.

RESTRUCTURED TEXT ft-rst-syntax

Syntax highlighting is enabled for code blocks within the document for a select number of file types. See $VIMRUNTIME/syntax/rst.vim for the default syntax list.

To set a user-defined list of code block syntax highlighting:

```
let rst_syntax_code_list = ['vim', 'lisp', ...]
```

To assign multiple code block types to a single syntax, define rst_syntax_code_list as a mapping:

```
let rst_syntax_code_list = {
        \ 'cpp': ['cpp', 'c++'],
        \ 'bash': ['bash', 'sh'],
        ...
\ }
```

To use color highlighting for emphasis text:

```
let rst_use_emphasis_colors = 1
```

To enable folding of sections:

```
let rst_fold_enabled = 1
```

Note that folding can cause performance issues on some platforms.

The minimum line syntax sync is set to 50. To modify this number:

```
let rst_minlines = 100
```

REXX ft-rexx-syntax

If you notice highlighting errors while scrolling backwards, which are fixed when redrawing with CTRL-L, try setting the "rexx_minlines" internal variable to a larger number:

```
:let rexx_minlines = 50
```

This will make the syntax synchronization start 50 lines before the first displayed line. The default value is 10. The disadvantage of using a larger number is that redrawing can become slow.

Vim tries to guess what type a ".r" file is. If it can't be detected (from comment lines), the default is "r". To make the default rexx add this line to your vimrc: g:filetype_r

```
:let g:filetype_r = "r"
```

RUBY ft-ruby-syntax

Ruby: Operator highlighting	ruby_operators
Ruby: Whitespace errors	ruby_space_errors
Ruby: Folding	ruby_fold ruby_foldable_groups

    Ruby: Reducing expensive operations ruby_no_expensive ruby_minlines
    Ruby: Spellchecking strings   ruby_spellcheck_strings

ruby_operators

Ruby: Operator highlighting

Operators can be highlighted by defining "ruby_operators":

```
:let ruby_operators = 1
```

ruby_space_errors

Ruby: Whitespace errors

Whitespace errors can be highlighted by defining "ruby_space_errors":

```
:let ruby_space_errors = 1
```

This will highlight trailing whitespace and tabs preceded by a space character as errors. This can be refined by defining "ruby_no_trail_space_error" and "ruby_no_tab_space_error" which will ignore trailing whitespace and tabs after spaces respectively.

ruby_fold ruby_foldable_groups

Ruby: Folding

Folding can be enabled by defining "ruby_fold":

```
:let ruby_fold = 1
```

This will set the value of 'foldmethod' to "syntax" locally to the current buffer or window, which will enable syntax-based folding when editing Ruby filetypes.

Default folding is rather detailed, i.e., small syntax units like "if", "do", "%w[]" may create corresponding fold levels.

You can set "ruby_foldable_groups" to restrict which groups are foldable:

```
:let ruby_foldable_groups = 'if case %'
```

The value is a space-separated list of keywords:

keyword	meaning
ALL	Most block syntax (default)
NONE	Nothing
if	"if" or "unless" block
def	"def" block
class	"class" block
module	"module" block
do	"do" block
begin	"begin" block
case	"case" block
for	"for", "while", "until" loops
{	Curly bracket block or hash literal
[	Array literal
%	Literal with "%" notation, e.g.: %w(STRING), %!STRING!
/	Regexp
string	String and shell command output (surrounded by ', ", "`")
:	Symbol
#	Multiline comment
<<	Here documents
__END__	Source code after "__END__" directive

ruby_no_expensive

Ruby: Reducing expensive operations

By default, the "end" keyword is colorized according to the opening statement of the block it closes. While useful, this feature can be expensive; if you experience slow redrawing (or you are on a terminal with poor color support) you may want to turn it off by defining the "ruby_no_expensive" variable:

```
:let ruby_no_expensive = 1
```

In this case the same color will be used for all control keywords.

ruby_minlines

If you do want this feature enabled, but notice highlighting errors while scrolling backwards, which are fixed when redrawing with CTRL-L, try setting the "ruby_minlines" variable to a value larger than 50:

```
:let ruby_minlines = 100
```

Ideally, this value should be a number of lines large enough to embrace your largest class or module.

ruby_spellcheck_strings

Ruby: Spellchecking strings

Ruby syntax will perform spellchecking of strings if you define "ruby_spellcheck_strings":

```
:let ruby_spellcheck_strings = 1
```

SCHEME ft-scheme-syntax

By default only R7RS keywords are highlighted and properly indented.

scheme.vim also supports extensions of the CHICKEN Scheme->C compiler.
Define b:is_chicken or g:is_chicken, if you need them.

SDL ft-sdl-syntax

The SDL highlighting probably misses a few keywords, but SDL has so many
of them it's almost impossibly to cope.

The new standard, SDL-2000, specifies that all identifiers are
case-sensitive (which was not so before), and that all keywords can be
used either completely lowercase or completely uppercase. To have the
highlighting reflect this, you can set the following variable:

```
:let sdl_2000=1
```

This also sets many new keywords. If you want to disable the old
keywords, which is probably a good idea, use:

```
:let SDL_no_96=1
```

The indentation is probably also incomplete, but right now I am very
satisfied with it for my own projects.

SED ft-sed-syntax

To make tabs stand out from regular blanks (accomplished by using Todo
highlighting on the tabs), define "g:sed_highlight_tabs" by putting

```
:let g:sed_highlight_tabs = 1
```

in the vimrc file. (This special highlighting only applies for tabs
inside search patterns, replacement texts, addresses or text included
by an Append/Change/Insert command.) If you enable this option, it is
also a good idea to set the tab width to one character; by doing that,
you can easily count the number of tabs in a string.

GNU sed allows comments after text on the same line. BSD sed only allows
comments where "#" is the first character of the line. To enforce BSD-style
comments, i.e. mark end-of-line comments as errors, use:

```
:let g:sed_dialect = "bsd"
```

Note that there are other differences between GNU sed and BSD sed which are
not (yet) affected by this setting.

Bugs:

  The transform command (y) is treated exactly like the substitute
  command. This means that, as far as this syntax file is concerned,
  transform accepts the same flags as substitute, which is wrong.
  (Transform accepts no flags.) I tolerate this bug because the
  involved commands need very complex treatment (95 patterns, one for
  each plausible pattern delimiter).

SGML ft-sgml-syntax

The coloring scheme for tags in the SGML file works as follows.

The <> of opening tags are colored differently than the </> of a closing tag.
This is on purpose! For opening tags the 'Function' color is used, while for
closing tags the 'Type' color is used (See syntax.vim to check how those are
defined for you)

Known tag names are colored the same way as statements in C. Unknown tag
names are not colored which makes it easy to spot errors.

Note that the same is true for argument (or attribute) names. Known attribute
names are colored differently than unknown ones.

Some SGML tags are used to change the rendering of text. The following tags
are recognized by the sgml.vim syntax coloring file and change the way normal
text is shown: <varname> <emphasis> <command> <function> <literal>
<replaceable> <ulink> and <link>.

If you want to change how such text is rendered, you must redefine the
following syntax groups:

- sgmlBold
- sgmlBoldItalic
- sgmlUnderline
- sgmlItalic

- sgmlLink for links

To make this redefinition work you must redefine them all and define the
following variable in your vimrc (this is due to the order in which the files
are read during initialization)

```
let sgml_my_rendering=1
```

You can also disable this rendering by adding the following line to your
vimrc file:

```
let sgml_no_rendering=1
```

(Adapted from the html.vim help text by Claudio Fleiner <claudio@fleiner.com>)

ft-posix-syntax ft-dash-syntax

SH ft-sh-syntax ft-bash-syntax ft-ksh-syntax

This covers syntax highlighting for the older Unix (Bourne) sh, and newer
shells such as bash, dash, posix, and the Korn shells.

Vim attempts to determine which shell type is in use by specifying that
various filenames are of specific types, e.g.:

```
ksh : .kshrc* *.ksh
bash: .bashrc* bashrc bash.bashrc .bash_profile* *.bash
```

See $VIMRUNTIME/filetype.vim for the full list of patterns. If none of these
cases pertain, then the first line of the file is examined (ex. looking for
/bin/sh /bin/ksh /bin/bash). If the first line specifies a shelltype, then
that shelltype is used. However some files (ex. .profile) are known to be
shell files but the type is not apparent. Furthermore, on many systems sh is
symbolically linked to "bash" (Linux, Windows+cygwin) or "ksh" (POSIX).

One may specify a global default by instantiating one of the following
variables in your vimrc:

   ksh:

```
let g:is_kornshell = 1
```

   posix: (default)

```
let g:is_posix     = 1
```

   bash:

```
let g:is_bash      = 1
```

   dash:

```
let g:is_dash      = 1
```

   sh: Bourne shell

```
let g:is_sh        = 1
```

Specific shell features are automatically enabled based on the shell detected
from the shebang line ("#! ...""). For KornShell Vim detects different shell
features for mksh, ksh88, ksh93, ksh93u, ksh93v, and ksh2020.

If there's no "#! ..." line, and the user hasn't availed themself of a default
sh.vim syntax setting as just shown, then syntax/sh.vim will assume the POSIX
shell syntax. No need to quote RFCs or market penetration statistics in error
reports, please -- just select the default version of the sh your system uses
and install the associated "let..." in your <.vimrc>.

The syntax/sh.vim file provides several levels of syntax-based folding:

```
let g:sh_fold_enabled= 0     (default, no syntax folding)
let g:sh_fold_enabled= 1     (enable function folding)
let g:sh_fold_enabled= 2     (enable heredoc folding)
let g:sh_fold_enabled= 4     (enable if/do/for folding)
```

then various syntax items (ie. HereDocuments and function bodies) become
syntax-foldable (see :syn-fold). You also may add these together
to get multiple types of folding:

```
let g:sh_fold_enabled= 3     (enables function and heredoc folding)
```

If you notice highlighting errors while scrolling backwards which are fixed
when one redraws with CTRL-L, try setting the "sh_minlines" internal variable
to a larger number. Example:

```
let sh_minlines = 500
```

This will make syntax synchronization start 500 lines before the first
displayed line. The default value is 200. The disadvantage of using a larger
number is that redrawing can become slow.

If you don't have much to synchronize on, displaying can be very slow. To
reduce this, the "sh_maxlines" internal variable can be set. Example:

```
let sh_maxlines = 100
```

The default is to use the twice sh_minlines. Set it to a smaller number to
speed up displaying. The disadvantage is that highlight errors may appear.

Unmatched "]", "done", "fi", etc., are highlighted as errors by default. If
you find this error handling problematic, you may suppress it with:

```
let g:sh_no_error = 1
```

As a refinement, you may suppress particular classes of errors by looking up
their syntax rule names and listing them with:

```
let g:sh_no_error_rules = ["shCurlyError", "shParenError"]
```

sh-embed sh-awk

Sh: EMBEDDING LANGUAGES~

You may wish to embed languages into sh. I'll give an example courtesy of
Lorance Stinson on how to do this with awk as an example. Put the following
file into $HOME/.config/nvim/after/syntax/sh/awkembed.vim:

```
" AWK Embedding:
" ==============
" Shamelessly ripped from aspperl.vim by Aaron Hope.
if exists("b:current_syntax")
  unlet b:current_syntax
endif
syn include @AWKScript syntax/awk.vim
syn region AWKScriptCode matchgroup=AWKCommand start=+[=\\]\@<!'+ skip=+\\'+ end=+'+ contains=@AWKScript contained
syn region AWKScriptEmbedded matchgroup=AWKCommand start=+\<awk\>+ skip=+\\$+ end=+[=\\]\@<!'+me=e-1 contains=@shIdList,@shExprList2
nextgroup=AWKScriptCode
syn cluster shCommandSubList add=AWKScriptEmbedded
hi def link AWKCommand Type
```

This code will then let the awk code in the single quotes:

```
awk '...awk code here...'
```

be highlighted using the awk highlighting syntax. Clearly this may be
extended to other languages.

SPEEDUP **ft-spup-syntax**

(AspenTech plant simulator)

The Speedup syntax file has some options:

- strict_subsections : If this variable is defined, only keywords for

  sections and subsections will be highlighted as statements but not
  other keywords (like WITHIN in the OPERATION section).
- highlight_types : Definition of this variable causes stream types

  like temperature or pressure to be highlighted as Type, not as a
  plain Identifier. Included are the types that are usually found in
  the DECLARE section; if you defined own types, you have to include
  them in the syntax file.
- oneline_comments : This value ranges from 1 to 3 and determines the

  highlighting of # style comments.

  oneline_comments = 1 : Allow normal Speedup code after an even
  number of #s.

  oneline_comments = 2 : Show code starting with the second # as
  error. This is the default setting.

  oneline_comments = 3 : Show the whole line as error if it contains
  more than one #.

Since especially OPERATION sections tend to become very large due to
PRESETting variables, syncing may be critical. If your computer is
fast enough, you can increase minlines and/or maxlines near the end of
the syntax file.

SQL **ft-sql-syntax**

ft-sqlinformix-syntax

ft-sqlanywhere-syntax

While there is an ANSI standard for SQL, most database engines add their own
custom extensions. Vim currently supports the Oracle and Informix dialects of

SQL. Vim assumes "*.sql" files are Oracle SQL by default.

Vim currently has SQL support for a variety of different vendors via syntax scripts. You can change Vim's default from Oracle to any of the current SQL supported types. You can also easily alter the SQL dialect being used on a buffer by buffer basis.

For more detailed instructions see ft_sql.txt.

SQUIRREL **ft-squirrel-syntax**

Squirrel is a high level imperative, object-oriented programming language, designed to be a light-weight scripting language that fits in the size, memory bandwidth, and real-time requirements of applications like video games. Files with the following extensions are recognized as squirrel files: .nut.

TCSH **ft-tcsh-syntax**

This covers the shell named "tcsh". It is a superset of csh. See ft-csh-syntax for how the filetype is detected.

Tcsh does not allow \" in strings unless the "backslash_quote" shell variable is set. If you want VIM to assume that no backslash quote constructs exist add this line to your vimrc:

```
:let tcsh_backslash_quote = 0
```

If you notice highlighting errors while scrolling backwards, which are fixed when redrawing with CTRL-L, try setting the "tcsh_minlines" internal variable to a larger number:

```
:let tcsh_minlines = 1000
```

This will make the syntax synchronization start 1000 lines before the first displayed line. If you set "tcsh_minlines" to "fromstart", then synchronization is done from the start of the file. The default value for tcsh_minlines is 100. The disadvantage of using a larger number is that redrawing can become slow.

TEX **ft-tex-syntax** **latex-syntax**

syntax-tex syntax-latex

Tex Contents~

Tex: Want Syntax Folding? tex-folding
Tex: No Spell Checking Wanted g:tex_nospell
Tex: Don't Want Spell Checking In Comments? tex-nospell
Tex: Want Spell Checking in Verbatim Zones? tex-verb
Tex: Run-on Comments or MathZones tex-runon
Tex: Slow Syntax Highlighting? tex-slow
Tex: Want To Highlight More Commands? tex-morecommands
Tex: Excessive Error Highlighting? tex-error
Tex: Need a new Math Group? tex-math
Tex: Starting a New Style? tex-style
Tex: Taking Advantage of Conceal Mode tex-conceal
Tex: Selective Conceal Mode g:tex_conceal
Tex: Controlling iskeyword g:tex_isk
Tex: Fine Subscript and Superscript Control tex-supersub
Tex: Match Check Control tex-matchcheck

tex-folding g:tex_fold_enabled

Tex: Want Syntax Folding?

As of version 28 of <syntax/tex.vim>, syntax-based folding of parts, chapters, sections, subsections, etc are supported. Put

```
let g:tex_fold_enabled=1
```

in your vimrc, and :set fdm=syntax. I suggest doing the latter via a modeline at the end of your LaTeX file:

```
% vim: fdm=syntax
```

If your system becomes too slow, then you might wish to look into

```
https://vimhelp.org/vim_faq.txt.html#faq-29.7
```

g:tex_nospell

Tex: No Spell Checking Wanted~

If you don't want spell checking anywhere in your LaTeX document, put

```
let g:tex_nospell=1
```

into your vimrc. If you merely wish to suppress spell checking inside comments only, see g:tex_comment_nospell.

tex-nospell g:tex_comment_nospell

Tex: Don't Want Spell Checking In Comments?

Some folks like to include things like source code in comments and so would
prefer that spell checking be disabled in comments in LaTeX files. To do
this, put the following in your vimrc:

```
let g:tex_comment_nospell= 1
```

If you want to suppress spell checking everywhere inside your LaTeX document,
see g:tex_nospell.

tex-verb g:tex_verbspell

Tex: Want Spell Checking in Verbatim Zones?~

Often verbatim regions are used for things like source code; seldom does
one want source code spell-checked. However, for those of you who do
want your verbatim zones spell-checked, put the following in your vimrc:

```
let g:tex_verbspell= 1
```

tex-runon tex-stopzone

Tex: Run-on Comments or MathZones

The <syntax/tex.vim> highlighting supports TeX, LaTeX, and some AmsTeX. The
highlighting supports three primary zones/regions: normal, texZone, and
texMathZone. Although considerable effort has been made to have these zones
terminate properly, zones delineated by $..$ and $$..$$ cannot be synchronized
as there's no difference between start and end patterns. Consequently, a
special "TeX comment" has been provided

```
%stopzone
```

which will forcibly terminate the highlighting of either a texZone or a
texMathZone.

tex-slow tex-sync

Tex: Slow Syntax Highlighting?

If you have a slow computer, you may wish to reduce the values for

```
:syn sync maxlines=200
:syn sync minlines=50
```

(especially the latter). If your computer is fast, you may wish to
increase them. This primarily affects synchronizing (i.e. just what group,
if any, is the text at the top of the screen supposed to be in?).

Another cause of slow highlighting is due to syntax-driven folding; see
tex-folding for a way around this.

g:tex_fast

Finally, if syntax highlighting is still too slow, you may set

```
:let g:tex_fast= ""
```

in your vimrc. Used this way, the g:tex_fast variable causes the syntax
highlighting script to avoid defining any regions and associated
synchronization. The result will be much faster syntax highlighting; the
price: you will no longer have as much highlighting or any syntax-based
folding, and you will be missing syntax-based error checking.

You may decide that some syntax is acceptable; you may use the following table
selectively to enable just some syntax highlighting:

```
b : allow bold and italic syntax
c : allow texComment syntax
m : allow texMatcher syntax (ie. {...} and [...])
M : allow texMath syntax
p : allow parts, chapter, section, etc syntax
r : allow texRefZone syntax (nocite, bibliography, label, pageref, eqref)
s : allow superscript/subscript regions
S : allow texStyle syntax
v : allow verbatim syntax
V : allow texNewEnv and texNewCmd syntax
```

As an example, let g:tex_fast= "M" will allow math-associated highlighting
but suppress all the other region-based syntax highlighting.
(also see: g:tex_conceal and tex-supersub)

tex-morecommands tex-package

Tex: Want To Highlight More Commands?

LaTeX is a programmable language, and so there are thousands of packages full
of specialized LaTeX commands, syntax, and fonts. If you're using such a
package you'll often wish that the distributed syntax/tex.vim would support
it. However, clearly this is impractical. So please consider using the
techniques in mysyntaxfile-add to extend or modify the highlighting provided
by syntax/tex.vim.

I've included some support for various popular packages on my website:

```
https://www.drchip.org/astronaut/vim/index.html#LATEXPKGS
```

The syntax files there go into your .../after/syntax/tex/ directory.

tex-error g:tex_no_error

Tex: Excessive Error Highlighting?

The <tex.vim> supports lexical error checking of various sorts. Thus, although the error checking is ofttimes very useful, it can indicate errors where none actually are. If this proves to be a problem for you, you may put in your vimrc the following statement:

```
let g:tex_no_error=1
```

and all error checking by <syntax/tex.vim> will be suppressed.

tex-math

Tex: Need a new Math Group?

If you want to include a new math group in your LaTeX, the following code shows you an example as to how you might do so:

```
call TexNewMathZone(sfx,mathzone,starform)
```

You'll want to provide the new math group with a unique suffix (currently, A-L and V-Z are taken by <syntax/tex.vim> itself). As an example, consider how eqnarray is set up by <syntax/tex.vim>:

```
call TexNewMathZone("D","eqnarray",1)
```

You'll need to change "mathzone" to the name of your new math group, and then to the call to it in .vim/after/syntax/tex.vim. The "starform" variable, if true, implies that your new math group has a starred form (ie. eqnarray*).

tex-style b:tex_stylish

Tex: Starting a New Style?

One may use "\makeatletter" in *.tex files, thereby making the use of "@" in commands available. However, since the *.tex file doesn't have one of the following suffices: sty cls clo dtx ltx, the syntax highlighting will flag such use of @ as an error. To solve this:

```
:let b:tex_stylish = 1
:set ft=tex
```

Putting "let g:tex_stylish=1" into your vimrc will make <syntax/tex.vim> always accept such use of @.

tex-cchar tex-cole tex-conceal

Tex: Taking Advantage of Conceal Mode~

If you have 'conceallevel' set to 2 and if your encoding is utf-8, then a number of character sequences can be translated into appropriate utf-8 glyphs, including various accented characters, Greek characters in MathZones, and superscripts and subscripts in MathZones. Not all characters can be made into superscripts or subscripts; the constraint is due to what utf-8 supports. In fact, only a few characters are supported as subscripts.

One way to use this is to have vertically split windows (see CTRL-W_v); one with 'conceallevel' at 0 and the other at 2; and both using 'scrollbind'.

g:tex_conceal

Tex: Selective Conceal Mode~

You may selectively use conceal mode by setting g:tex_conceal in your vimrc. By default, g:tex_conceal is set to "admgs" to enable concealment for the following sets of characters:

```
a = accents/ligatures
b = bold and italic
d = delimiters
m = math symbols
g = Greek
s = superscripts/subscripts
```

By leaving one or more of these out, the associated conceal-character substitution will not be made.

g:tex_isk g:tex_stylish

Tex: Controlling iskeyword~

Normally, LaTeX keywords support 0-9, a-z, A-z, and 192-255 only. Latex keywords don't support the underscore - except when in *.sty files. The syntax highlighting script handles this with the following logic:

```
        * If g:tex_stylish exists and is 1
                then the file will be treated as a "sty" file, so the "_"
                will be allowed as part of keywords
                (regardless of g:tex_isk)
        * Else if the file's suffix is sty, cls, clo, dtx, or ltx,
                then the file will be treated as a "sty" file, so the "_"
                will be allowed as part of keywords
                (regardless of g:tex_isk)
```

* If g:tex_isk exists, then it will be used for the local 'iskeyword'
* Else the local 'iskeyword' will be set to 48-57,a-z,A-Z,192-255

tex-supersub g:tex_superscripts g:tex_subscripts

Tex: Fine Subscript and Superscript Control~

See tex-conceal for how to enable concealed character replacement.

See g:tex_conceal for selectively concealing accents, bold/italic, math, Greek, and superscripts/subscripts.

One may exert fine control over which superscripts and subscripts one wants syntax-based concealment for (see :syn-cchar). Since not all fonts support all characters, one may override the concealed-replacement lists; by default these lists are given by:

```
let g:tex_superscripts= "[0-9a-zA-W.,:;+-<>/()=]"
let g:tex_subscripts= "[0-9aehijklmnoprstuvx,+-/().]"
```

For example, I use Luxi Mono Bold; it doesn't support subscript characters for "hklmnpst", so I put

```
let g:tex_subscripts= "[0-9aeijoruvx,+-/().]"
```

in ~/.config/nvim/ftplugin/tex/tex.vim in order to avoid having inscrutable utf-8 glyphs appear.

tex-matchcheck g:tex_matchcheck

Tex: Match Check Control~

Sometimes one actually wants mismatched parentheses, square braces, and or curly braces; for example, \text{(1,10]} is a range from but not including 1 to and including 10. This wish, of course, conflicts with the desire to provide delimiter mismatch detection. To accommodate these conflicting goals, syntax/tex.vim provides

```
g:tex_matchcheck = '[({[]'
```

which is shown along with its default setting. So, if one doesn't want [] and () to be checked for mismatches, try using

```
let g:tex_matchcheck= '[{}]'
```

If you don't want matching to occur inside bold and italicized regions,

```
let g:tex_excludematcher= 1
```

will prevent the texMatcher group from being included in those regions.

TF ft-tf-syntax

There is one option for the tf syntax highlighting.

For syncing, minlines defaults to 100. If you prefer another value, you can set "tf_minlines" to the value you desire. Example:

```
:let tf_minlines = your choice
```

TYPESCRIPT ft-typescript-syntax ft-typescriptreact-syntax

There is one option to control the TypeScript syntax highlighting.

g:typescript_host_keyword

When this variable is set to 1, host-specific APIs such as addEventListener are highlighted. To disable set it to zero in your .vimrc:

```
let g:typescript_host_keyword = 0
```

The default value is 1.

TYPST ft-typst-syntax

g:typst_embedded_languages

Typst files can embed syntax highlighting for other languages by setting the g:typst_embedded_languages variable. This variable is a list of language names whose syntax definitions will be included in Typst files. Example:

```
let g:typst_embedded_languages = ['python', 'r']
```

VIM ft-vim-syntax g:vimsyn_minlines g:vimsyn_maxlines

There is a trade-off between more accurate syntax highlighting versus screen updating speed. To improve accuracy, you may wish to increase the g:vimsyn_minlines variable. The g:vimsyn_maxlines variable may be used to improve screen updating rates (see :syn-sync for more on this).

```
g:vimsyn_minlines : used to set synchronization minlines
g:vimsyn_maxlines : used to set synchronization maxlines
```

(g:vim_minlines and g:vim_maxlines are deprecated variants of
these two options)

g:vimsyn_embed

The g:vimsyn_embed option allows users to select what, if any, types of
embedded script highlighting they wish to have.

```
g:vimsyn_embed == 0       : disable (don't embed any scripts)
g:vimsyn_embed ==# 'lpPr' : support embedded Lua, Perl, Python and Ruby
```

By default, g:vimsyn_embed is unset, and embedded Lua scripts are supported.

g:vimsyn_folding

Some folding is now supported with when 'foldmethod' is set to "syntax":

```
g:vimsyn_folding == 0 or doesn't exist: no syntax-based folding
g:vimsyn_folding =~# 'a' : fold augroups
g:vimsyn_folding =~# 'f' : fold functions
g:vimsyn_folding =~# 'h' : fold let heredocs
g:vimsyn_folding =~# 'l' : fold Lua      heredocs
g:vimsyn_folding =~# 'p' : fold Perl     heredocs
g:vimsyn_folding =~# 'P' : fold Python   heredocs
g:vimsyn_folding =~# 'r' : fold Ruby     heredocs
```

By default, g:vimsyn_folding is unset. Concatenate the indicated characters
to support folding of multiple syntax constructs (e.g.,
g:vimsyn_folding = "fh" will enable folding of both functions and heredocs).

g:vimsyn_comment_strings

By default, strings are highlighted inside comments. This may be disabled by
setting g:vimsyn_comment_strings to false.

g:vimsyn_noerror

Not all error highlighting that syntax/vim.vim does may be correct; Vim script
is a difficult language to highlight correctly. A way to suppress error
highlighting is to put the following line in your vimrc:

```
let g:vimsyn_noerror = 1
```

WDL **wdl-syntax**

The Workflow Description Language is a way to specify data processing
workflows with a human-readable and writeable syntax. This is used a lot in
bioinformatics. More info on the spec can be found here:
https://github.com/openwdl/wdl

XF86CONFIG **ft-xf86conf-syntax**

The syntax of XF86Config file differs in XFree86 v3.x and v4.x. Both
variants are supported. Automatic detection is used, but is far from perfect.
You may need to specify the version manually. Set the variable
xf86conf_xfree86_version to 3 or 4 according to your XFree86 version in
your vimrc. Example:

```
:let xf86conf_xfree86_version=3
```

When using a mix of versions, set the b:xf86conf_xfree86_version variable.

Note that spaces and underscores in option names are not supported. Use
"SyncOnGreen" instead of "__s yn con gr_e_e_n" if you want the option name
highlighted.

XML **ft-xml-syntax**

Xml namespaces are highlighted by default. This can be inhibited by
setting a global variable:

```
:let g:xml_namespace_transparent=1
```

xml-folding

The xml syntax file provides syntax folding (see :syn-fold) between
start and end tags. This can be turned on by

```
:let g:xml_syntax_folding = 1
:set foldmethod=syntax
```

Note: Syntax folding might slow down syntax highlighting significantly,
especially for large files.

X Pixmaps (XPM) ft-xpm-syntax

xpm.vim creates its syntax items dynamically based upon the contents of the
XPM file. Thus if you make changes e.g. in the color specification strings,
you have to source it again e.g. with ":set syn=xpm".

To copy a pixel with one of the colors, yank a "pixel" with "yl" and insert it
somewhere else with "P".

Do you want to draw with the mouse? Try the following:

```
:function! GetPixel()
:   let c = getline(".")[col(".") - 1]
:   echo c
:   exe "noremap <LeftMouse> <LeftMouse>r" .. c
:   exe "noremap <LeftDrag>     <LeftMouse>r" .. c
:endfunction
:noremap <RightMouse> <LeftMouse>:call GetPixel()<CR>
:set guicursor=n:hor20     " to see the color beneath the cursor
```

This turns the right button into a pipette and the left button into a pen.
It will work with XPM files that have one character per pixel only and you
must not click outside of the pixel strings, but feel free to improve it.

It will look much better with a font in a quadratic cell size, e.g. for X:

```
:set guifont=-*-clean-medium-r-*-*-8-*-*-*-*-80-*
```

YAML ft-yaml-syntax

g:yaml_schema b:yaml_schema

A YAML schema is a combination of a set of tags and a mechanism for resolving
non-specific tags. For user this means that YAML parser may, depending on
plain scalar contents, treat plain scalar (which can actually be only string
and nothing else) as a value of the other type: null, boolean, floating-point,
integer. g:yaml_schema option determines according to which schema values
will be highlighted specially. Supported schemas are

Schema	Description
failsafe	No additional highlighting.
json	Supports JSON-style numbers, booleans and null.
core	Supports more number, boolean and null styles.
pyyaml	In addition to core schema supports highlighting timestamps, but there are some differences in what is recognized as numbers and many additional boolean values not present in core schema.

Default schema is core.

Note that schemas are not actually limited to plain scalars, but this is the
only difference between schemas defined in YAML specification and the only
difference defined in the syntax file.

ZSH ft-zsh-syntax

The syntax script for zsh allows for syntax-based folding:

```
:let g:zsh_fold_enable = 1
```

6. Defining a syntax :syn-define E410

Vim understands three types of syntax items:

1. Keyword

   It can only contain keyword characters, according to the characters
   specified with :syn-iskeyword or the 'iskeyword' option. It cannot
   contain other syntax items. It will only match with a complete word (there
   are no keyword characters before or after the match). The keyword "if"
   would match in "if(a=b)", but not in "ifdef x", because "(" is not a
   keyword character and "d" is.
2. Match

   This is a match with a single regexp pattern.
3. Region

   This starts at a match of the "start" regexp pattern and ends with a match
   with the "end" regexp pattern. Any other text can appear in between. A
   "skip" regexp pattern can be used to avoid matching the "end" pattern.

Several syntax ITEMs can be put into one syntax GROUP. For a syntax group
you can give highlighting attributes. For example, you could have an item
to define a /* ... */ comment and another one that defines a "// ..."
comment, and put them both in the "Comment" group. You can then specify that
a "Comment" will be in bold font and have a blue color. You are free to make
one highlight group for one syntax item, or put all items into one group.
This depends on how you want to specify your highlighting attributes. Putting
each item in its own group results in having to specify the highlighting
for a lot of groups.

Note that a syntax group and a highlight group are similar. For a highlight
group you will have given highlight attributes. These attributes will be used
for the syntax group with the same name.

In case more than one item matches at the same position, the one that was defined LAST wins. Thus you can override previously defined syntax items by using an item that matches the same text. But a keyword always goes before a match or region. And a keyword with matching case always goes before a keyword with ignoring case.

PRIORITY :syn-priority

When several syntax items may match, these rules are used:

1. When multiple Match or Region items start in the same position, the item defined last has priority.
2. A Keyword has priority over Match and Region items.
3. An item that starts in an earlier position has priority over items that start in later positions.

DEFINING CASE :syn-case E390

:sy[ntax] case [match | ignore]
This defines if the following ":syntax" commands will work with matching case, when using "match", or with ignoring case, when using "ignore". Note that any items before this are not affected, and all items until the next ":syntax case" command are affected.

:sy[ntax] case
Show either "syntax case match" or "syntax case ignore".

DEFINING FOLDLEVEL :syn-foldlevel

:sy[ntax] foldlevel start
:sy[ntax] foldlevel minimum
This defines how the foldlevel of a line is computed when using foldmethod=syntax (see fold-syntax and :syn-fold):

start: Use level of item containing start of line.
minimum: Use lowest local-minimum level of items on line.

The default is "start". Use "minimum" to search a line horizontally for the lowest level contained on the line that is followed by a higher level. This produces more natural folds when syntax items may close and open horizontally within a line.

:sy[ntax] foldlevel
Show the current foldlevel method, either "syntax foldlevel start" or "syntax foldlevel minimum".

SPELL CHECKING :syn-spell

:sy[ntax] spell toplevel
:sy[ntax] spell notoplevel
:sy[ntax] spell default
This defines where spell checking is to be done for text that is not in a syntax item:

toplevel: Text is spell checked.
notoplevel: Text is not spell checked.
default: When there is a @Spell cluster no spell checking.

For text in syntax items use the @Spell and @NoSpell clusters spell-syntax. When there is no @Spell and no @NoSpell cluster then spell checking is done for "default" and "toplevel".

To activate spell checking the 'spell' option must be set.

:sy[ntax] spell
Show the current syntax spell checking method, either "syntax spell toplevel", "syntax spell notoplevel" or "syntax spell default".

SYNTAX ISKEYWORD SETTING :syn-iskeyword

:sy[ntax] iskeyword [clear | {option}]
This defines the keyword characters. It's like the 'iskeyword' option for but only applies to syntax highlighting.

clear: Syntax specific iskeyword setting is disabled and the buffer-local 'iskeyword' setting is used.
{option} Set the syntax 'iskeyword' option to a new value.

Example:

```
:syntax iskeyword @,48-57,192-255,$,_
```

This would set the syntax specific iskeyword option to include all alphabetic characters, plus the numeric characters, all accented characters and also includes the "_" and the "$".

If no argument is given, the current value will be output.

Setting this option influences what /\k matches in syntax patterns and also determines where :syn-keyword will be checked for a new match.

It is recommended when writing syntax files, to use this command to
set the correct value for the specific syntax language and not change
the 'iskeyword' option.

DEFINING KEYWORDS :syn-keyword

:sy[ntax] keyword {group-name} [{options}] {keyword} ... [{options}]

This defines a number of keywords.

{group-name}   Is a syntax group name such as "Comment".
[{options}]   See :syn-arguments below.
{keyword} ...   Is a list of keywords which are part of this group.

Example:

```
:syntax keyword   Type   int long char
```

The {options} can be given anywhere in the line. They will apply to
all keywords given, also for options that come after a keyword.
These examples do exactly the same:

```
:syntax keyword   Type   contained int long char
:syntax keyword   Type   int long contained char
:syntax keyword   Type   int long char contained
```

E789 E890

When you have a keyword with an optional tail, like Ex commands in
Vim, you can put the optional characters inside [], to define all the
variations at once:

```
:syntax keyword   vimCommand     ab[breviate] n[ext]
```

Don't forget that a keyword can only be recognized if all the
characters are included in the 'iskeyword' option. If one character
isn't, the keyword will never be recognized.
Multi-byte characters can also be used. These do not have to be in
'iskeyword'.
See :syn-iskeyword for defining syntax specific iskeyword settings.

A keyword always has higher priority than a match or region, the
keyword is used if more than one item matches. Keywords do not nest
and a keyword can't contain anything else.

Note that when you have a keyword that is the same as an option (even
one that isn't allowed here), you can not use it. Use a match
instead.

The maximum length of a keyword is 80 characters.

The same keyword can be defined multiple times, when its containment
differs. For example, you can define the keyword once not contained
and use one highlight group, and once contained, and use a different
highlight group. Example:

```
:syn keyword vimCommand tag
:syn keyword vimSetting contained tag
```

When finding "tag" outside of any syntax item, the "vimCommand"
highlight group is used. When finding "tag" in a syntax item that
contains "vimSetting", the "vimSetting" group is used.

DEFINING MATCHES :syn-match

:sy[ntax] match {group-name} [{options}]
                [excludenl]
                [keepend]
                {pattern}
                [{options}]

This defines one match.

{group-name}   A syntax group name such as "Comment".
[{options}]   See :syn-arguments below.
[excludenl]   Don't make a pattern with the end-of-line "$"
                        extend a containing match or region. Must be
                        given before the pattern. :syn-excludenl
keepend   Don't allow contained matches to go past a
                        match with the end pattern. See
                        :syn-keepend.
{pattern}   The search pattern that defines the match.
                        See :syn-pattern below.
                        Note that the pattern may match more than one
                        line, which makes the match depend on where
                        Vim starts searching for the pattern. You
                        need to make sure syncing takes care of this.

Example (match a character constant):

```
:syntax match Character /'.'/hs=s+1,he=e-1
```

DEFINING REGIONS :syn-region :syn-start :syn-skip :syn-end

E398 E399

:sy[ntax] region {group-name} [{options}]
                [matchgroup={group-name}]

```
        [keepend]
        [extend]
        [excludenl]
        start={start-pattern} ...
        [skip={skip-pattern}]
        end={end-pattern} ...
        [{options}]
```

This defines one region. It may span several lines.

{group-name}	A syntax group name such as "Comment".
[{options}]	See :syn-arguments below.
[matchgroup={group-name}]	The syntax group to use for the following start or end pattern matches only. Not used for the text in between the matched start and end patterns. Use NONE to reset to not using a different group for the start or end match. See :syn-matchgroup.
keepend	Don't allow contained matches to go past a match with the end pattern. See :syn-keepend.
extend	Override a "keepend" for an item this region is contained in. See :syn-extend.
excludenl	Don't make a pattern with the end-of-line "$" extend a containing match or item. Only useful for end patterns. Must be given before the patterns it applies to. :syn-excludenl
start={start-pattern}	The search pattern that defines the start of the region. See :syn-pattern below.
skip={skip-pattern}	The search pattern that defines text inside the region where not to look for the end pattern. See :syn-pattern below.
end={end-pattern}	The search pattern that defines the end of the region. See :syn-pattern below.

Example:

```
:syntax region String   start=+"+  skip=+\\"+  end=+"+
```

The start/skip/end patterns and the options can be given in any order. There can be zero or one skip pattern. There must be one or more start and end patterns. This means that you can omit the skip pattern, but you must give at least one start and one end pattern. It is allowed to have white space before and after the equal sign (although it mostly looks better without white space).

When more than one start pattern is given, a match with one of these is sufficient. This means there is an OR relation between the start patterns. The last one that matches is used. The same is true for the end patterns.

The search for the end pattern starts right after the start pattern. Offsets are not used for this. This implies that the match for the end pattern will never overlap with the start pattern.

The skip and end pattern can match across line breaks, but since the search for the pattern can start in any line it often does not do what you want. The skip pattern doesn't avoid a match of an end pattern in the next line. Use single-line patterns to avoid trouble.

Note: The decision to start a region is only based on a matching start pattern. There is no check for a matching end pattern. This does NOT work:

```
:syn region First  start="("  end=":"
:syn region Second start="("  end=";"
```

The Second always matches before the First (last defined pattern has higher priority). The Second region then continues until the next ';', no matter if there is a ':' before it. Using a match does work:

```
:syn match First  "(\_.\{-}:"
:syn match Second "(\_.\{-};"
```

This pattern matches any character or line break with "_." and repeats that with "\{-}" (repeat as few as possible).

:syn-keepend

By default, a contained match can obscure a match for the end pattern. This is useful for nesting. For example, a region that starts with "{" and ends with "}", can contain another region. An encountered "}" will then end the contained region, but not the outer region:

```
    {           starts outer "{}" region
        {       starts contained "{}" region
        }       ends contained "{}" region
    }           ends outer "{} region
```

If you don't want this, the "keepend" argument will make the matching of an end pattern of the outer region also end any contained item. This makes it impossible to nest the same region, but allows for contained items to highlight parts of the end pattern, without causing that to skip the match with the end pattern. Example:

```
:syn match  vimComment +"[^"]\+$+
:syn region vimCommand start="set" end="$" contains=vimComment keepend
```

The "keepend" makes the vimCommand always end at the end of the line,
even though the contained vimComment includes a match with the <EOL>.

When "keepend" is not used, a match with an end pattern is retried
after each contained match. When "keepend" is included, the first
encountered match with an end pattern is used, truncating any
contained matches.

:syn-extend

The "keepend" behavior can be changed by using the "extend" argument.
When an item with "extend" is contained in an item that uses
"keepend", the "keepend" is ignored and the containing region will be
extended.
This can be used to have some contained items extend a region while
others don't. Example:

```
:syn region htmlRef start=+<a>+ end=+</a>+ keepend contains=htmlItem,htmlScript
:syn match htmlItem +<[^>]*>+ contained
:syn region htmlScript start=+<script+ end=+</script[^>]*>+ contained extend
```

Here the htmlItem item does not make the htmlRef item continue
further, it is only used to highlight the <> items. The htmlScript
item does extend the htmlRef item.

Another example:

```
:syn region xmlFold start="<a>" end="</a>" fold transparent keepend extend
```

This defines a region with "keepend", so that its end cannot be
changed by contained items, like when the "" is matched to
highlight it differently. But when the xmlFold region is nested (it
includes itself), the "extend" applies, so that the "" of a nested
region only ends that region, and not the one it is contained in.

:syn-excludenl

When a pattern for a match or end pattern of a region includes a '$'
to match the end-of-line, it will make a region item that it is
contained in continue on the next line. For example, a match with
"\\$" (backslash at the end of the line) can make a region continue
that would normally stop at the end of the line. This is the default
behavior. If this is not wanted, there are two ways to avoid it:

1. Use "keepend" for the containing item. This will keep all
   contained matches from extending the match or region. It can be
   used when all contained items must not extend the containing item.
2. Use "excludenl" in the contained item. This will keep that match
   from extending the containing match or region. It can be used if
   only some contained items must not extend the containing item.
   "excludenl" must be given before the pattern it applies to.

:syn-matchgroup

"matchgroup" can be used to highlight the start and/or end pattern
differently than the body of the region. Example:

```
:syntax region String matchgroup=Quote start=+"+  skip=+\\"+    end=+"+
```

This will highlight the quotes with the "Quote" group, and the text in
between with the "String" group.
The "matchgroup" is used for all start and end patterns that follow,
until the next "matchgroup". Use "matchgroup=NONE" to go back to not
using a matchgroup.

In a start or end pattern that is highlighted with "matchgroup" the
contained items of the region are not used. This can be used to avoid
that a contained item matches in the start or end pattern match. When
using "transparent", this does not apply to a start or end pattern
match that is highlighted with "matchgroup".

Here is an example, which highlights three levels of parentheses in
different colors:

```
:sy region par1 matchgroup=par1 start=/(/ end=/)/ contains=par2
:sy region par2 matchgroup=par2 start=/(/ end=/)/ contains=par3 contained
:sy region par3 matchgroup=par3 start=/(/ end=/)/ contains=par1 contained
:hi par1 ctermfg=red guifg=red
:hi par2 ctermfg=blue guifg=blue
:hi par3 ctermfg=darkgreen guifg=darkgreen
```

E849

The maximum number of syntax groups is 19999.

7. :syntax arguments :syn-arguments

The :syntax commands that define syntax items take a number of arguments.
The common ones are explained here. The arguments may be given in any order
and may be mixed with patterns.

Not all commands accept all arguments. This table shows which arguments
can not be used for all commands:

E395

	contains	oneline	fold	display	extend	concealends~
:syntax keyword	-	-	-	-	-	-
:syntax match	yes	-	yes	yes	yes	-
:syntax region	yes	yes	yes	yes	yes	yes

These arguments can be used for all three commands:

- conceal
- cchar
- contained
- containedin
- nextgroup
- transparent
- skipwhite
- skipnl
- skipempty

conceal conceal :syn-conceal

When the "conceal" argument is given, the item is marked as concealable.
Whether or not it is actually concealed depends on the value of the
'conceallevel' option. The 'concealcursor' option is used to decide whether
concealable items in the current line are displayed unconcealed to be able to
edit the line.

Another way to conceal text is with matchadd(), but internally this works a
bit differently syntax-vs-match.

concealends :syn-concealends

When the "concealends" argument is given, the start and end matches of
the region, but not the contents of the region, are marked as concealable.
Whether or not they are actually concealed depends on the setting on the
'conceallevel' option. The ends of a region can only be concealed separately
in this way when they have their own highlighting via "matchgroup". The
synconcealed() function can be used to retrieve information about concealed
items.

cchar :syn-cchar

E844

The "cchar" argument defines the character shown in place of the item
when it is concealed (setting "cchar" only makes sense when the conceal
argument is given.) If "cchar" is not set then the default conceal
character defined in the 'listchars' option is used. The character cannot be
a control character such as Tab. Example:

```
:syntax match Entity "&amp;" conceal cchar=&
```

See hl-Conceal for highlighting.

contained :syn-contained

When the "contained" argument is given, this item will not be recognized at
the top level, but only when it is mentioned in the "contains" field of
another match. Example:

```
:syntax keyword Todo    TODO    contained
:syntax match   Comment "//.*"  contains=Todo
```

display :syn-display

If the "display" argument is given, this item will be skipped when the
detected highlighting will not be displayed. This will speed up highlighting,
by skipping this item when only finding the syntax state for the text that is
to be displayed.

Generally, you can use "display" for match and region items that meet these
conditions:

- The item does not continue past the end of a line. Example for C: A region
  for a "/*" comment can't contain "display", because it continues on the next
  line.
- The item does not contain items that continue past the end of the line or
  make it continue on the next line.
- The item does not change the size of any item it is contained in. Example
  for C: A match with "\\$" in a preprocessor match can't have "display",
  because it may make that preprocessor match shorter.

- The item does not allow other items to match that didn't match otherwise,

  and that item may extend the match too far. Example for C: A match for a
  "//" comment can't use "display", because a "/*" inside that comment would
  match then and start a comment which extends past the end of the line.

Examples, for the C language, where "display" can be used:
- match with a number

- match with a label

transparent :syn-transparent

If the "transparent" argument is given, this item will not be highlighted
itself, but will take the highlighting of the item it is contained in. This
is useful for syntax items that don't need any highlighting but are used
only to skip over a part of the text.

The "contains=" argument is also inherited from the item it is contained in,
unless a "contains" argument is given for the transparent item itself. To
avoid that unwanted items are contained, use "contains=NONE". Example, which
highlights words in strings, but makes an exception for "vim":

```
:syn match myString /'[^']*'/ contains=myWord,myVim
:syn match myWord   /\<[a-z]*\>/ contained
:syn match myVim    /\<vim\>/ transparent contained contains=NONE
:hi link myString String
:hi link myWord   Comment
```

Since the "myVim" match comes after "myWord" it is the preferred match (last
match in the same position overrules an earlier one). The "transparent"
argument makes the "myVim" match use the same highlighting as "myString". But
it does not contain anything. If the "contains=NONE" argument would be left
out, then "myVim" would use the contains argument from myString and allow
"myWord" to be contained, which will be highlighted as a Comment. This
happens because a contained match doesn't match inside itself in the same
position, thus the "myVim" match doesn't overrule the "myWord" match here.

When you look at the colored text, it is like looking at layers of contained
items. The contained item is on top of the item it is contained in, thus you
see the contained item. When a contained item is transparent, you can look
through, thus you see the item it is contained in. In a picture:

```
               look from here

           |   |   |   |   |   |
           V   V   V   V   V   V

              xxxx       yyy          more contained items
           ...................        contained item (transparent)
       ============================   first item
```

The 'x', 'y' and '=' represent a highlighted syntax item. The '.' represent a
transparent group.

What you see is:

```
       =======xxxx=======yyy========
```

Thus you look through the transparent "....".

oneline :syn-oneline

The "oneline" argument indicates that the region does not cross a line
boundary. It must match completely in the current line. However, when the
region has a contained item that does cross a line boundary, it continues on
the next line anyway. A contained item can be used to recognize a line
continuation pattern. But the "end" pattern must still match in the first
line, otherwise the region doesn't even start.

When the start pattern includes a "\n" to match an end-of-line, the end
pattern must be found in the same line as where the start pattern ends. The
end pattern may also include an end-of-line. Thus the "oneline" argument
means that the end of the start pattern and the start of the end pattern must
be within one line. This can't be changed by a skip pattern that matches a
line break.

fold :syn-fold

The "fold" argument makes the fold level increase by one for this item.
Example:

```
:syn region myFold start="{" end="}" transparent fold
:syn sync fromstart
:set foldmethod=syntax
```

This will make each {} block form one fold.

The fold will start on the line where the item starts, and end where the item
ends. If the start and end are within the same line, there is no fold.
The 'foldnestmax' option limits the nesting of syntax folds.
See :syn-foldlevel to control how the foldlevel of a line is computed
from its syntax items.

:syn-contains E405 E406 E407 E408 E409

contains={group-name},...

The "contains" argument is followed by a list of syntax group names. These groups will be allowed to begin inside the item (they may extend past the containing group's end). This allows for recursive nesting of matches and regions. If there is no "contains" argument, no groups will be contained in this item. The group names do not need to be defined before they can be used here.

contains=ALL
If the only item in the contains list is "ALL", then all groups will be accepted inside the item.

contains=ALLBUT,{group-name},...
If the first item in the contains list is "ALLBUT", then all groups will be accepted inside the item, except the ones that are listed. Example:

```
:syntax region Block start="{" end="}" ... contains=ALLBUT,Function
```

contains=TOP
If the first item in the contains list is "TOP", then all groups will be accepted that don't have the "contained" argument.
contains=TOP,{group-name},...
Like "TOP", but excluding the groups that are listed.

contains=CONTAINED
If the first item in the contains list is "CONTAINED", then all groups will be accepted that have the "contained" argument.
contains=CONTAINED,{group-name},...
Like "CONTAINED", but excluding the groups that are listed.

The {group-name} in the "contains" list can be a pattern. All group names that match the pattern will be included (or excluded, if "ALLBUT" is used). The pattern cannot contain white space or a ','. Example:

```
... contains=Comment.*,Keyw[0-3]
```

The matching will be done at moment the syntax command is executed. Groups that are defined later will not be matched. Also, if the current syntax command defines a new group, it is not matched. Be careful: When putting syntax commands in a file you can't rely on groups NOT being defined, because the file may have been sourced before, and ":syn clear" doesn't remove the group names.

The contained groups will also match in the start and end patterns of a region. If this is not wanted, the "matchgroup" argument can be used :syn-matchgroup. The "ms=" and "me=" offsets can be used to change the region where contained items do match. Note that this may also limit the area that is highlighted

containedin={group-name},... :syn-containedin

The "containedin" argument is followed by a list of syntax group names. The item will be allowed to begin inside these groups. This works as if the containing item has a "contains=" argument that includes this item.

Only the immediate containing item (the one at the top of the syntax stack) is considered. Vim does not search other ancestors. If the immediate container neither contains this item via :syn-contains nor is named in this item's "containedin=", the match will not start even if some ancestor would allow it. Note that a :syn-transparent region still enforces its own :syn-contains list.

The {group-name},... can be used just like for "contains", as explained above.

This is useful when adding a syntax item afterwards. An item can be told to be included inside an already existing item, without changing the definition of that item. For example, to highlight a word in a C comment after loading the C syntax:

```
:syn keyword myword HELP containedin=cComment contained
```

Note that "contained" is also used, to avoid that the item matches at the top level.

Matches for "containedin" are added to the other places where the item can appear. A "contains" argument may also be added as usual. Don't forget that keywords never contain another item, thus adding them to "containedin" won't work.
See also: :syn-contains, :syn-transparent.

nextgroup={group-name},... :syn-nextgroup

The "nextgroup" argument is followed by a list of syntax group names, separated by commas (just like with "contains", so you can also use patterns).

If the "nextgroup" argument is given, the mentioned syntax groups will be tried for a match, after the match or region ends. If none of the groups have a match, highlighting continues normally. If there is a match, this group will be used, even when it is not mentioned in the "contains" field of the

current group. This is like giving the mentioned group priority over all
other groups. Example:

```
:syntax match  ccFoobar  "Foo.\{-}Bar"  contains=ccFoo
:syntax match  ccFoo     "Foo"      contained nextgroup=ccFiller
:syntax region ccFiller  start="."  matchgroup=ccBar  end="Bar"  contained
```

This will highlight "Foo" and "Bar" differently, and only when there is a
"Bar" after "Foo". In the text line below, "f" shows where ccFoo is used for
highlighting, and "bbb" where ccBar is used.

```
Foo asdfasd Bar asdf Foo asdf Bar asdf
fff            bbb      fff      bbb
```

Note the use of ".\{-}" to skip as little as possible until the next Bar.
when .* would be used, the "asdf" in between "Bar" and "Foo" would be
highlighted according to the "ccFoobar" group, because the ccFooBar match
would include the first "Foo" and the last "Bar" in the line (see pattern).

skipwhite :syn-skipwhite

skipnl :syn-skipnl

skipempty :syn-skipempty

These arguments are only used in combination with "nextgroup". They can be
used to allow the next group to match after skipping some text:
        skipwhite   skip over space and tab characters
        skipnl   skip over the end of a line
        skipempty   skip over empty lines (implies a "skipnl")

When "skipwhite" is present, the white space is only skipped if there is no
next group that matches the white space.

When "skipnl" is present, the match with nextgroup may be found in the next
line. This only happens when the current item ends at the end of the current
line! When "skipnl" is not present, the nextgroup will only be found after
the current item in the same line.

When skipping text while looking for a next group, the matches for other
groups are ignored. Only when no next group matches, other items are tried
for a match again. This means that matching a next group and skipping white
space and <EOL>s has a higher priority than other items.

Example:

```
:syn match ifstart "\<if.*"       nextgroup=ifline skipwhite skipempty
:syn match ifline  "[^ \t].*" nextgroup=ifline skipwhite skipempty contained
:syn match ifline  "endif"      contained
```

Note that the [^ \t].* match matches all non-white text. Thus it would also
match "endif". Therefore the "endif" match is put last, so that it takes
precedence.
Note that this example doesn't work for nested "if"s. You need to add
"contains" arguments to make that work (omitted for simplicity of the
example).

IMPLICIT CONCEAL :syn-conceal-implicit

:sy[ntax] conceal [on|off]
        This defines if the following ":syntax" commands will define keywords,
        matches or regions with the "conceal" flag set. After ":syn conceal
        on", all subsequent ":syn keyword", ":syn match" or ":syn region"
        defined will have the "conceal" flag set implicitly. ":syn conceal
        off" returns to the normal state where the "conceal" flag must be
        given explicitly.

:sy[ntax] conceal
        Show either "syntax conceal on" or "syntax conceal off".

8. Syntax patterns :syn-pattern E401 E402

In the syntax commands, a pattern must be surrounded by two identical
characters. This is like it works for the ":s" command. The most common to
use is the double quote. But if the pattern contains a double quote, you can
use another character that is not used in the pattern. Examples:

```
:syntax region Comment  start="/\*"  end="\*/"
:syntax region String   start=+"+    end=+"+     skip=+\\"+
```

See pattern for the explanation of what a pattern is. Syntax patterns are
always interpreted like the 'magic' option is set, no matter what the actual
value of 'magic' is. And the patterns are interpreted like the 'l' flag is
not included in 'cpoptions'. This was done to make syntax files portable and
independent of the 'magic' setting.

Try to avoid patterns that can match an empty string, such as "[a-z]*".
This slows down the highlighting a lot, because it matches everywhere.

:syn-pattern-offset

The pattern can be followed by a character offset. This can be used to
change the highlighted part, and to change the text area included in the
match or region (which only matters when trying to match other items). Both
are relative to the matched pattern. The character offset for a skip
pattern can be used to tell where to continue looking for an end pattern.

The offset takes the form of "{what}={offset}"
The {what} can be one of seven strings:

```
ms      Match Start     offset for the start of the matched text
me      Match End       offset for the end of the matched text
hs      Highlight Start offset for where the highlighting starts
he      Highlight End   offset for where the highlighting ends
rs      Region Start    offset for where the body of a region starts
re      Region End      offset for where the body of a region ends
lc      Leading Context offset past "leading context" of pattern
```

The {offset} can be:

```
s       start of the matched pattern
s+{nr}  start of the matched pattern plus {nr} chars to the right
s-{nr}  start of the matched pattern plus {nr} chars to the left
e       end of the matched pattern
e+{nr}  end of the matched pattern plus {nr} chars to the right
e-{nr}  end of the matched pattern plus {nr} chars to the left
{nr}    (for "lc" only): start matching {nr} chars right of the start
```

Examples: "ms=s+1", "hs=e-2", "lc=3".

Although all offsets are accepted after any pattern, they are not always
meaningful. This table shows which offsets are actually used:

ms	me hs he	rs re	lc
match item	yes yes yes yes	- -	yes
region item start yes - yes -	yes -	yes	
region item skip -	yes - -	- -	yes
region item end -	yes - yes	- yes yes	

Offsets can be concatenated, with a ',' in between. Example:

```
:syn match String  /"[^"]*"/hs=s+1,he=e-1
```

```
    some "string" text
         ^^^^^^               highlighted
```

Notes:

- There must be no white space between the pattern and the character

  offset(s).
- The highlighted area will never be outside of the matched text.

- A negative offset for an end pattern may not always work, because the end

  pattern may be detected when the highlighting should already have stopped.
- Before Vim 7.2 the offsets were counted in bytes instead of characters.

  This didn't work well for multibyte characters, so it was changed with the
  Vim 7.2 release.
- The start of a match cannot be in a line other than where the pattern

  matched. This doesn't work: "a\nb"ms=e. You can make the highlighting
  start in another line, this does work: "a\nb"hs=e.

Example (match a comment but don't highlight the /* and */):

```
:syntax region Comment start="/\*"hs=e+1 end="\*/"he=s-1
```

```
/* this is a comment */
  ^^^^^^^^^^^^^^^^^^     highlighted
```

A more complicated Example:

```
:syn region Exa matchgroup=Foo start="foo"hs=s+2,rs=e+2 matchgroup=Bar end="bar"me=e-1,he=e-1,re=s-1
```

```
abcfoostringbarabc
   mmmmmmmmmmm      match
     sssrrreee      highlight start/region/end ("Foo", "Exa" and "Bar")
```

Leading context :syn-lc :syn-leading :syn-context

Note: This is an obsolete feature, only included for backwards compatibility
with previous Vim versions. It's now recommended to use the /\@<= construct
in the pattern. You can also often use /\zs.

The "lc" offset specifies leading context -- a part of the pattern that must
be present, but is not considered part of the match. An offset of "lc=n" will
cause Vim to step back n columns before attempting the pattern match, allowing
characters which have already been matched in previous patterns to also be

used as leading context for this match. This can be used, for instance, to specify that an "escaping" character must not precede the match:

```
:syn match ZNoBackslash "[^\\]z"ms=s+1
:syn match WNoBackslash "[^\\]w"lc=1
:syn match Underline "_\+"
```

```
         ___zzzz ___wwww
         ^^^     ^^^     matches Underline
             ^ ^         matches ZNoBackslash
                   ^^^^ matches WNoBackslash
```

The "ms" offset is automatically set to the same value as the "lc" offset, unless you set "ms" explicitly.

Multi-line patterns :syn-multi-line

The patterns can include "\n" to match an end-of-line. Mostly this works as expected, but there are a few exceptions.

When using a start pattern with an offset, the start of the match is not allowed to start in a following line. The highlighting can start in a following line though. Using the "\zs" item also requires that the start of the match doesn't move to another line.

The skip pattern can include the "\n", but the search for an end pattern will continue in the first character of the next line, also when that character is matched by the skip pattern. This is because redrawing may start in any line halfway in a region and there is no check if the skip pattern started in a previous line. For example, if the skip pattern is "a\nb" and an end pattern is "b", the end pattern does match in the second line of this:

```
x x a
b x x
```

Generally this means that the skip pattern should not match any characters after the "\n".

External matches :syn-ext-match

These extra regular expression items are available in region patterns:

                                    /\z(/\z(\) E50 E52 E879
    \z(\)   Marks the sub-expression as "external", meaning that it can be
                accessed from another pattern match. Currently only usable in
                defining a syntax region start pattern.

                                    /\z1 /\z2 /\z3 /\z4 /\z5
    \z1 ... \z9   /\z6 /\z7 /\z8 /\z9 E66 E67
                Matches the same string that was matched by the corresponding
                sub-expression in a previous start pattern match.

Sometimes the start and end patterns of a region need to share a common sub-expression. A common example is the "here" document in Perl and many Unix shells. This effect can be achieved with the "\z" special regular expression items, which marks a sub-expression as "external", in the sense that it can be referenced from outside the pattern in which it is defined. The here-document example, for instance, can be done like this:

```
:syn region hereDoc start="<<\z(\I\i*\)" end="^\z1$"
```

As can be seen here, the \z actually does double duty. In the start pattern, it marks the "\(\I\i*\)" sub-expression as external; in the end pattern, it changes the \z1 back-reference into an external reference referring to the first external sub-expression in the start pattern. External references can also be used in skip patterns:

```
:syn region foo start="start \z(\I\i*\)" skip="not end \z1" end="end \z1"
```

Note that normal and external sub-expressions are completely orthogonal and indexed separately; for instance, if the pattern "\z(..\)\(..\)" is applied to the string "aabb", then \1 will refer to "bb" and \z1 will refer to "aa". Note also that external sub-expressions cannot be accessed as back-references within the same pattern like normal sub-expressions. If you want to use one sub-expression as both a normal and an external sub-expression, you can nest the two, as in "\(\z(...\)\)".

Note that only matches within a single line can be used. Multi-line matches cannot be referred to.

9. Syntax clusters :syn-cluster E400

```
:sy[ntax] cluster {cluster-name} [contains={group-name},...]
                                 [add={group-name},...]
                                 [remove={group-name},...]
```

This command allows you to cluster a list of syntax groups together under a single name.

        contains={group-name},...
                The cluster is set to the specified list of groups.
        add={group-name},...
                The specified groups are added to the cluster.

remove={group-name},...
The specified groups are removed from the cluster.

A cluster so defined may be referred to in a contains=..., containedin=..., nextgroup=..., add=... or remove=... list with a "@" prefix. You can also use this notation to implicitly declare a cluster before specifying its contents.

Example:

```
:syntax match Thing "# [^#]\+ #" contains=@ThingMembers
:syntax cluster ThingMembers contains=ThingMember1,ThingMember2
```

As the previous example suggests, modifications to a cluster are effectively retroactive; the membership of the cluster is checked at the last minute, so to speak:

```
:syntax keyword A aaa
:syntax keyword B bbb
:syntax cluster AandB contains=A
:syntax match Stuff "( aaa bbb )" contains=@AandB
:syntax cluster AandB add=B       " now both keywords are matched in Stuff
```

This also has implications for nested clusters:

```
:syntax keyword A aaa
:syntax keyword B bbb
:syntax cluster SmallGroup contains=B
:syntax cluster BigGroup contains=A,@SmallGroup
:syntax match Stuff "( aaa bbb )" contains=@BigGroup
:syntax cluster BigGroup remove=B       " no effect, since B isn't in BigGroup
:syntax cluster SmallGroup remove=B     " now bbb isn't matched within Stuff
```

E848

The maximum number of clusters is 9767.

10. Including syntax files

:syn-include E397

It is often useful for one language's syntax file to include a syntax file for a related language. Depending on the exact relationship, this can be done in two different ways:

- If top-level syntax items in the included syntax file are to be

  allowed at the top level in the including syntax, you can simply use the :runtime command:

```
" In cpp.vim:
:runtime! syntax/c.vim
:unlet b:current_syntax
```

- If top-level syntax items in the included syntax file are to be

  contained within a region in the including syntax, you can use the ":syntax include" command:

:sy[ntax] include [@{grouplist-name}] {file-name}

All syntax items declared in the included file will have the "contained" flag added. In addition, if a group list is specified, all top-level syntax items in the included file will be added to that list.

```
" In perl.vim:
:syntax include @Pod <script>:p:h/pod.vim
:syntax region perlPOD start="^=head" end="^=cut" contains=@Pod
```

When {file-name} is an absolute path (starts with "/", "c:", "$VAR" or "<script>") that file is sourced. When it is a relative path (e.g., "syntax/pod.vim") the file is searched for in 'runtimepath'. All matching files are loaded. Using a relative path is recommended, because it allows a user to replace the included file with their own version, without replacing the file that does the ":syn include".

E847

The maximum number of includes is 999.

11. Synchronizing

:syn-sync E403 E404

Vim wants to be able to start redrawing in any position in the document. To make this possible it needs to know the syntax state at the position where redrawing starts.

:sy[ntax] sync [ccomment [group-name] | minlines={N} | ...]

There are four ways to synchronize:

1. Always parse from the start of the file.

   :syn-sync-first
2. Based on C-style comments. Vim understands how C-comments work and can

   figure out if the current line starts inside or outside a comment.
   :syn-sync-second
3. Jumping back a certain number of lines and start parsing there.

   :syn-sync-third
4. Searching backwards in the text for a pattern to sync on.

   :syn-sync-fourth

:syn-sync-maxlines :syn-sync-minlines

For the last three methods, the line range where the parsing can start is limited by "minlines" and "maxlines".

If the "minlines={N}" argument is given, the parsing always starts at least that many lines backwards. This can be used if the parsing may take a few lines before it's correct, or when it's not possible to use syncing.

If the "maxlines={N}" argument is given, the number of lines that are searched for a comment or syncing pattern is restricted to N lines backwards (after adding "minlines"). This is useful if you have few things to sync on and a slow machine. Example:

```
:syntax sync maxlines=500 ccomment
```

:syn-sync-linebreaks

When using a pattern that matches multiple lines, a change in one line may cause a pattern to no longer match in a previous line. This means has to start above where the change was made. How many lines can be specified with the "linebreaks" argument. For example, when a pattern may include one line break use this:

```
:syntax sync linebreaks=1
```

The result is that redrawing always starts at least one line before where a change was made. The default value for "linebreaks" is zero. Usually the value for "minlines" is bigger than "linebreaks".

First syncing method: :syn-sync-first

```
:syntax sync fromstart
```

The file will be parsed from the start. This makes syntax highlighting accurate, but can be slow for long files. Vim caches previously parsed text, so that it's only slow when parsing the text for the first time. However, when making changes some part of the text needs to be parsed again (worst case: to the end of the file).

Using "fromstart" is equivalent to using "minlines" with a very large number.

Second syncing method: :syn-sync-second :syn-sync-ccomment

For the second method, only the "ccomment" argument needs to be given. Example:

```
:syntax sync ccomment
```

When Vim finds that the line where displaying starts is inside a C-style comment, the last region syntax item with the group-name "Comment" will be used. This requires that there is a region with the group-name "Comment"! An alternate group name can be specified, for example:

```
:syntax sync ccomment javaComment
```

This means that the last item specified with "syn region javaComment" will be used for the detected C comment region. This only works properly if that region does have a start pattern "\/*" and an end pattern "*\/".

The "maxlines" argument can be used to restrict the search to a number of lines. The "minlines" argument can be used to at least start a number of lines back (e.g., for when there is some construct that only takes a few lines, but it hard to sync on).

Note: Syncing on a C comment doesn't work properly when strings are used that cross a line and contain a "*/". Since letting strings cross a line is a bad programming habit (many compilers give a warning message), and the chance of a "*/" appearing inside a comment is very small, this restriction is hardly ever noticed.

Third syncing method: :syn-sync-third

For the third method, only the "minlines={N}" argument needs to be given.
Vim will subtract {N} from the line number and start parsing there. This
means {N} extra lines need to be parsed, which makes this method a bit slower.
Example:

```
:syntax sync minlines=50
```

"lines" is equivalent to "minlines" (used by older versions).

Fourth syncing method: :syn-sync-fourth

The idea is to synchronize on the end of a few specific regions, called a
sync pattern. Only regions can cross lines, so when we find the end of some
region, we might be able to know in which syntax item we are. The search
starts in the line just above the one where redrawing starts. From there
the search continues backwards in the file.

This works just like the non-syncing syntax items. You can use contained
matches, nextgroup, etc. But there are a few differences:

- Keywords cannot be used.
- The syntax items with the "sync" keyword form a completely separated group of syntax items. You can't mix syncing groups and non-syncing groups.
- The matching works backwards in the buffer (line by line), instead of forwards.
- A line continuation pattern can be given. It is used to decide which group of lines need to be searched like they were one line. This means that the search for a match with the specified items starts in the first of the consecutive lines that contain the continuation pattern.
- When using "nextgroup" or "contains", this only works within one line (or group of continued lines).
- When using a region, it must start and end in the same line (or group of continued lines). Otherwise the end is assumed to be at the end of the line (or group of continued lines).
- When a match with a sync pattern is found, the rest of the line (or group of continued lines) is searched for another match. The last match is used. This is used when a line can contain both the start and the end of a region (e.g., in a C-comment like /* this */, the last "*/" is used).

There are two ways how a match with a sync pattern can be used:

1. Parsing for highlighting starts where redrawing starts (and where the search for the sync pattern started). The syntax group that is expected to be valid there must be specified. This works well when the regions that cross lines cannot contain other regions.
2. Parsing for highlighting continues just after the match. The syntax group that is expected to be present just after the match must be specified. This can be used when the previous method doesn't work well. It's much slower, because more text needs to be parsed.

Both types of sync patterns can be used at the same time.

Besides the sync patterns, other matches and regions can be specified, to
avoid finding unwanted matches.

[The reason that the sync patterns are given separately, is that mostly the
search for the sync point can be much simpler than figuring out the
highlighting. The reduced number of patterns means it will go (much)
faster.]

syn-sync-grouphere E393 E394

:syntax sync match {sync-group-name} grouphere {group-name} "pattern" ...

Define a match that is used for syncing. {group-name} is the
name of a syntax group that follows just after the match. Parsing
of the text for highlighting starts just after the match. A region
must exist for this {group-name}. The first one defined will be used.
"NONE" can be used for when there is no syntax group after the match.

syn-sync-groupthere

:syntax sync match {sync-group-name} groupthere {group-name} "pattern" ...

Like "grouphere", but {group-name} is the name of a syntax group that
is to be used at the start of the line where searching for the sync
point started. The text between the match and the start of the sync
pattern searching is assumed not to change the syntax highlighting.
For example, in C you could search backwards for "/*" and "*/". If
"/*" is found first, you know that you are inside a comment, so the

"groupthere" is "cComment". If "*/" is found first, you know that you are not in a comment, so the "groupthere" is "NONE". (in practice it's a bit more complicated, because the "/*" and "*/" could appear inside a string. That's left as an exercise to the reader...).

:syntax sync match ...
:syntax sync region ...

Without a "groupthere" argument. Define a region or match that is skipped while searching for a sync point.

syn-sync-linecont

:syntax sync linecont {pattern}

When {pattern} matches in a line, it is considered to continue in the next line. This means that the search for a sync point will consider the lines to be concatenated.

If the "maxlines={N}" argument is given too, the number of lines that are searched for a match is restricted to N. This is useful if you have very few things to sync on and a slow machine. Example:

```
:syntax sync maxlines=100
```

You can clear all sync settings with:

```
:syntax sync clear
```

You can clear specific sync patterns with:

```
:syntax sync clear {sync-group-name} ...
```

12. Listing syntax items

:syntax :sy :syn :syn-list

This command lists all the syntax items:

```
:sy[ntax] [list]
```

To show the syntax items for one syntax group:

```
:sy[ntax] list {group-name}
```

To list the syntax groups in one cluster: E392

```
:sy[ntax] list @{cluster-name}
```

See above for other arguments for the ":syntax" command.

Note that the ":syntax" command can be abbreviated to ":sy", although ":syn" is mostly used, because it looks better.

13. Highlight command

:highlight :hi E28 E411 E415

Nvim uses a range of highlight groups which fall into two categories: Editor interface and syntax highlighting. In rough order of importance, these are

- basic editor highlight-groups
- standard syntax group-names (in addition, syntax files can define language-specific groups, which are prefixed with the language name)
- diagnostic-highlights
- treesitter-highlight-groups
- lsp-semantic-highlight groups
- lsp-highlight of symbols and references

Where appropriate, highlight groups are linked by default to one of the more basic groups, but colorschemes are expected to cover all of them. Under each tag, the corresponding highlight groups are highlighted using the current colorscheme.

:colo :colorscheme E185

:colo[rscheme] Output the name of the currently active color scheme. This is basically the same as

```
:echo g:colors_name
```

In case g:colors_name has not been defined :colo will
output "default".

:colo[rscheme] {name} Load color scheme {name}. This searches 'runtimepath'
for the file "colors/{name}.{vim,lua}". The first one
that is found is loaded.
Note: "colors/{name}.vim" is tried first.
Also searches all plugins in 'packpath', first below
"start" and then under "opt".

Doesn't work recursively, thus you can't use
":colorscheme" in a color scheme script.

To customize a color scheme use another name, e.g.
"~/.config/nvim/colors/mine.vim", and use :runtime to
load the original color scheme:

```
runtime colors/evening.vim
hi Statement ctermfg=Blue guifg=Blue
```

Before the color scheme will be loaded the
ColorSchemePre autocommand event is triggered.
After the color scheme has been loaded the
ColorScheme autocommand event is triggered.
For info about writing a color scheme file:

```
:edit $VIMRUNTIME/colors/README.txt
```

:hi[ghlight] List all the current highlight groups that have
attributes set.

:hi[ghlight] {group-name}
List one highlight group.

highlight-clear :hi-clear

:hi[ghlight] clear Reset all highlighting to the defaults. Removes all
highlighting for groups added by the user.
Uses the current value of 'background' to decide which
default colors to use.
If there was a default link, restore it. :hi-link

:hi[ghlight] clear {group-name}
:hi[ghlight] {group-name} NONE
Disable the highlighting for one highlight group. It
is _not_ set back to the default colors.

:hi[ghlight] [default] {group-name} {key}={arg} ...
Add a highlight group, or change the highlighting for
an existing group.
See highlight-args for the {key}={arg} arguments.
See :highlight-default for the optional [default]
argument.

Normally a highlight group is added once when starting up. This sets the
default values for the highlighting. After that, you can use additional
highlight commands to change the arguments that you want to set to non-default
values. The value "NONE" can be used to switch the value off or go back to
the default value.

A simple way to change colors is with the :colorscheme command. This loads
a file with ":highlight" commands such as this:

```
:hi Comment     gui=bold
```

Note that all settings that are not included remain the same, only the
specified field is used, and settings are merged with previous ones. So, the
result is like this single command has been used:

```
:hi Comment     ctermfg=Cyan guifg=#80a0ff gui=bold
```

:highlight-verbose

When listing a highlight group and 'verbose' is non-zero, the listing will
also tell where it was last set. Example:

```
:verbose hi Comment
```

Comment   xxx ctermfg=4 guifg=Blue
Last set from /home/mool/vim/vim7/runtime/syntax/syncolor.vim

When ":hi clear" is used then the script where this command is used will be
mentioned for the default values. See :verbose-cmd for more information.

highlight-args E416 E417 E423

There are two types of UIs for highlighting:
cterm   terminal UI (TUI)
gui   GUI or RGB-capable TUI ('termguicolors')

For each type the highlighting can be given. This makes it possible to use
the same syntax file on all UIs.

1. TUI highlight arguments

bold underline undercurl
underdouble underdotted
underdashed inverse italic
standout strikethrough altfont
dim blink hl-conceal overline nocombine

cterm={attr-list} attr-list highlight-cterm E418

attr-list is a comma-separated list (without spaces) of the following items (in any order):

bold	
underline	
undercurl	curly underline
underdouble	double underline
underdotted	dotted underline
underdashed	dashed underline
strikethrough	
reverse	
inverse	same as reverse
italic	
standout	
altfont	
dim	half-bright/faint text
blink	blinking text
conceal	concealed/hidden text
overline	overlined text
nocombine	override attributes instead of combining them
NONE	no attributes used (used to reset it)

Note that "bold" can be used here and by using a bold font. They have the same effect.
"undercurl", "underdouble", "underdotted", and "underdashed" fall back to "underline" in a terminal that does not support them. The color is set using guisp.

start={term-list} highlight-start E422
stop={term-list} term-list highlight-stop

These lists of terminal codes can be used to get non-standard attributes on a terminal.

The escape sequence specified with the "start" argument is written before the characters in the highlighted area. It can be anything that you want to send to the terminal to highlight this area. The escape sequence specified with the "stop" argument is written after the highlighted area. This should undo the "start" argument. Otherwise the screen will look messed up.

{term-list} is a string with escape sequences. This is any string of characters, except that it can't start with "t_" and blanks are not allowed. The <> notation is recognized here, so you can use things like "<Esc>" and "<Space>". Example:

```
start=<Esc>[27h;<Esc>[<Space>r;
```

ctermfg={color-nr} ctermfg E421
ctermbg={color-nr} ctermbg

The {color-nr} argument is a color number. Its range is zero to (not including) the number of tui-colors available.
The actual color with this number depends on the type of terminal and its settings. Sometimes the color also depends on the settings of "cterm". For example, on some systems "cterm=bold ctermfg=3" gives another color, on others you just get color 3.

The following (case-insensitive) names are recognized:

cterm-colors

NR-16	NR-8	COLOR NAME
0	0	Black
1	4	DarkBlue
2	2	DarkGreen
3	6	DarkCyan
4	1	DarkRed
5	5	DarkMagenta
6	3	Brown, DarkYellow
7	7	LightGray, LightGrey, Gray, Grey
8	0*	DarkGray, DarkGrey
9	4*	Blue, LightBlue
10	2*	Green, LightGreen
11	6*	Cyan, LightCyan
12	1*	Red, LightRed
13	5*	Magenta, LightMagenta
14	3*	Yellow, LightYellow
15	7*	White

The number under "NR-16" is used for 16-color terminals ('t_Co' greater than or equal to 16). The number under "NR-8" is used for 8-color terminals ('t_Co' less than 16). The "*" indicates that the bold attribute is set for ctermfg. In many 8-color terminals (e.g., "linux"), this causes the bright colors to appear. This doesn't work for background colors! Without the "*" the bold attribute is removed. If you want to set the bold attribute in a different way, put a "cterm=" argument AFTER the "ctermfg=" or "ctermbg=" argument. Or use a number instead of a color name.

Note that for 16 color ansi style terminals (including xterms), the
numbers in the NR-8 column is used. Here "*" means "add 8" so that
Blue is 12, DarkGray is 8 etc.

Note that for some color terminals these names may result in the wrong
colors!

You can also use "NONE" to remove the color.

:hi-normal-cterm

When setting the "ctermfg" or "ctermbg" colors for the Normal group,
these will become the colors used for the non-highlighted text.
Example:

```
:highlight Normal ctermfg=grey ctermbg=darkblue
```

When setting the "ctermbg" color for the Normal group, the
'background' option will be adjusted automatically, under the
condition that the color is recognized and 'background' was not set
explicitly. This causes the highlight groups that depend on
'background' to change! This means you should set the colors for
Normal first, before setting other colors.
When a color scheme is being used, changing 'background' causes it to
be reloaded, which may reset all colors (including Normal). First
delete the "g:colors_name" variable when you don't want this.

When you have set "ctermfg" or "ctermbg" for the Normal group, Vim
needs to reset the color when exiting. This is done with the
"orig_pair" terminfo entry.
E419 E420
When Vim knows the normal foreground and background colors, "fg" and
"bg" can be used as color names. This only works after setting the
colors for the Normal group and for the MS-Windows console. Example,
for reverse video:

```
:highlight Visual ctermfg=bg ctermbg=fg
```

Note that the colors are used that are valid at the moment this
command are given. If the Normal group colors are changed later, the
"fg" and "bg" colors will not be adjusted.

1. GUI highlight arguments

gui={attr-list} highlight-gui

These give the attributes to use in the GUI mode.
See attr-list for a description.
Note that "bold" can be used here and by using a bold font. They
have the same effect.
Note that the attributes are ignored for the "Normal" group.

font={font-name} highlight-font

font-name is the name of a font, as it is used on the system Vim
runs on. For X11 this is a complicated name, for example:

```
font=-misc-fixed-bold-r-normal--14-130-75-75-c-70-iso8859-1
```

The font-name "NONE" can be used to revert to the default font.
When setting the font for the "Normal" group, this becomes the default
font (until the 'guifont' option is changed; the last one set is
used).
The following only works with Motif not with other GUIs:
When setting the font for the "Menu" group, the menus will be changed.
When setting the font for the "Tooltip" group, the tooltips will be
changed.
All fonts used, except for Menu and Tooltip, should be of the same
character size as the default font! Otherwise redrawing problems will
occur.
To use a font name with an embedded space or other special character,
put it in single quotes. The single quote cannot be used then.
Example:

```
:hi comment font='Monospace 10'
```

guifg={color-name} guifg

guibg={color-name} guibg

guisp={color-name} guisp

These give the foreground (guifg), background (guibg) and special
(guisp) color to use in the GUI. "guisp" is used for various
underlines.
There are a few special names:

NONE	no color (transparent)
bg	use normal background color
background	use normal background color
fg	use normal foreground color
foreground	use normal foreground color

To use a color name with an embedded space or other special character,
put it in single quotes. The single quote cannot be used then.
Example:

```
:hi comment guifg='salmon pink'
```

gui-colors

Suggested color names (these are available on most systems):

Red	LightRed	DarkRed	
Green	LightGreen	DarkGreen	SeaGreen
Blue	LightBlue	DarkBlue	SlateBlue
Cyan	LightCyan	DarkCyan	
Magenta	LightMagenta	DarkMagenta	
Yellow	LightYellow	Brown	DarkYellow
Gray	LightGray	DarkGray	
Black	White		
Orange	Purple	Violet	

Colors which define Nvim's default color scheme:

NvimDarkBlue	NvimLightBlue
NvimDarkCyan	NvimLightCyan
NvimDarkGray1	NvimLightGray1
NvimDarkGray2	NvimLightGray2
NvimDarkGray3	NvimLightGray3
NvimDarkGray4	NvimLightGray4
NvimDarkGreen	NvimLightGreen
NvimDarkMagenta	NvimLightMagenta
NvimDarkRed	NvimLightRed
NvimDarkYellow	NvimLightYellow

You can also specify a color by its RGB (red, green, blue) values.
The format is "#rrggbb", where
        "rr"   is the Red value
        "gg"   is the Green value
        "bb"   is the Blue value
All values are hexadecimal, range from "00" to "ff". Examples:

```
:highlight Comment guifg=#11f0c3 guibg=#ff00ff
```

blend={integer} highlight-blend opacity

Override the blend level for a highlight group within the popupmenu
or floating windows. Only takes effect if 'pumblend' or 'winblend'
is set for the menu or window. See the help at the respective option.

See also the "blend" flag of nvim_buf_set_extmark().

highlight-groups highlight-default

These are the builtin highlighting groups. Note that the highlighting depends
on the value of 'background'. You can see the current settings with the
":highlight" command.

hl-ColorColumn

ColorColumn Used for the columns set with 'colorcolumn'.

hl-Conceal

Conceal Placeholder characters substituted for concealed
text (see 'conceallevel').

hl-CurSearch

CurSearch Current match for the last search pattern (see 'hlsearch').
Note: This is correct after a search, but may get outdated if
changes are made or the screen is redrawn.

hl-Cursor hl-lCursor

Cursor Character under the cursor.
lCursor Character under the cursor when language-mapping
is used (see 'guicursor').

hl-CursorIM

CursorIM Like Cursor, but used when in IME mode. CursorIM

hl-CursorColumn

CursorColumn Screen-column at the cursor, when 'cursorcolumn' is set.

hl-CursorLine

CursorLine Screen-line at the cursor, when 'cursorline' is set.
Low-priority if foreground (ctermfg OR guifg) is not set.

hl-Directory

Directory Directory names (and other special names in listings).

hl-DiffAdd

DiffAdd Diff mode: Added line. diff.txt

hl-DiffChange

DiffChange Diff mode: Changed line. diff.txt

hl-DiffDelete

DiffDelete Diff mode: Deleted line. diff.txt

hl-DiffText

DiffText Diff mode: Changed text within a changed line. diff.txt

hl-DiffTextAdd

DiffTextAdd Diff mode: Added text within a changed line. Linked to hl-DiffText by default. diff.txt

hl-EndOfBuffer

EndOfBuffer Filler lines (~) after the last line in the buffer. By default, this is highlighted like hl-NonText.

hl-TermCursor

TermCursor Cursor in a focused terminal.

hl-OkMsg

OkMsg Success messages.

hl-WarningMsg

WarningMsg Warning messages.

hl-ErrorMsg

ErrorMsg Error messages.

hl-StderrMsg

StderrMsg Messages in stderr from shell commands.

hl-StdoutMsg

StdoutMsg Messages in stdout from shell commands.

hl-WinSeparator

WinSeparator Separators between window splits.

hl-Folded

Folded Line used for closed folds.

hl-FoldColumn

FoldColumn 'foldcolumn'

hl-SignColumn

SignColumn Column where signs are displayed.

hl-IncSearch

IncSearch 'incsearch' highlighting; also used for the text replaced with ":s///c".

hl-Substitute

Substitute :substitute replacement text highlighting.

hl-LineNr

LineNr Line number for ":number" and ":#" commands, and when 'number' or 'relativenumber' option is set.

hl-LineNrAbove

LineNrAbove Line number for when the 'relativenumber' option is set, above the cursor line.

hl-LineNrBelow

LineNrBelow Line number for when the 'relativenumber' option is set, below the cursor line.

hl-CursorLineNr

CursorLineNr Like LineNr when 'cursorline' is set and 'cursorlineopt' contains "number" or is "both", for the cursor line.

hl-CursorLineFold

CursorLineFold Like FoldColumn when 'cursorline' is set for the cursor line.

hl-CursorLineSign

CursorLineSign Like SignColumn when 'cursorline' is set for the cursor line.

hl-MatchParen

MatchParen Character under the cursor or just before it, if it is a paired bracket, and its match. pi_paren.txt

hl-MCursor

MCursor multicursor cursor.

hl-MCursorVisual

MCursorVisual multicursor Visual selection.

hl-ModeMsg

ModeMsg 'showmode' message (e.g., "-- INSERT --").

hl-MsgArea

MsgArea Area for messages and command-line, see also 'cmdheight'.

hl-MsgSeparator

MsgSeparator Separator for scrolled messages msgsep.

hl-MoreMsg

MoreMsg more-prompt

hl-NonText

NonText '@' at the end of the window, characters from 'showbreak' and other characters that do not really exist in the text (e.g., ">" displayed when a double-wide character doesn't fit at the end of the line). See also hl-EndOfBuffer.

hl-Normal

Normal Normal text.

hl-NormalFloat

NormalFloat Normal text in floating windows.

hl-FloatBorder

FloatBorder Border of floating windows.

hl-FloatShadow

FloatShadow Blended areas when border is "shadow".

hl-FloatShadowThrough

FloatShadowThrough Shadow corners when border is "shadow".

hl-FloatTitle

FloatTitle Title of floating windows.

hl-FloatFooter

FloatFooter Footer of floating windows.

hl-NormalNC

NormalNC Normal text in non-current windows.

hl-Pmenu

Pmenu Popup menu: Normal item.

hl-PmenuSel

PmenuSel Popup menu: Selected item. Combined with hl-Pmenu.

hl-PmenuKind

PmenuKind Popup menu: Normal item "kind".

hl-PmenuKindSel

PmenuKindSel Popup menu: Selected item "kind".

hl-PmenuExtra

PmenuExtra Popup menu: Normal item "extra text".

hl-PmenuExtraSel

PmenuExtraSel Popup menu: Selected item "extra text".

hl-PmenuSbar

PmenuSbar Popup menu: Scrollbar.

hl-PmenuThumb

PmenuThumb Popup menu: Thumb of the scrollbar.

hl-PmenuMatch

PmenuMatch Popup menu: Matched text in normal item. Combined with hl-Pmenu.

hl-PmenuMatchSel

PmenuMatchSel Popup menu: Matched text in selected item. Combined with hl-PmenuMatch and hl-PmenuSel.

hl-PmenuBorder

PmenuBorder Popup menu: border of popup menu.

hl-PmenuShadow

PmenuShadow Popup menu: blended areas when 'pumborder' is "shadow".

hl-PmenuShadowThrough

PmenuShadowThrough Popup menu: shadow corners when 'pumborder' is "shadow".

hl-ComplMatchIns

ComplMatchIns Matched text of the currently inserted completion.

hl-PreInsert

PreInsert Text inserted when "preinsert" is in 'completeopt'.

hl-ComplHint

ComplHint Virtual text of the currently selected completion.

hl-ComplHintMore

ComplHintMore The additional information of the virtual text.

hl-Question

Question hit-enter prompt and yes/no questions.

hl-QuickFixLine

QuickFixLine Current quickfix item in the quickfix window. Combined with

hl-CursorLine when the cursor is there.

hl-Search

Search Last search pattern highlighting (see 'hlsearch').
Also used for similar items that need to stand out.

hl-SnippetTabstop

SnippetTabstop Tabstops in snippets. vim.snippet

hl-SnippetTabstopActive

SnippetTabstopActive
The currently active tabstop. vim.snippet

hl-SpecialKey

SpecialKey Unprintable characters: Text displayed differently from what
it really is. But not 'listchars' whitespace. hl-Whitespace

hl-SpellBad

SpellBad Word that is not recognized by the spellchecker. spell
Combined with the highlighting used otherwise.

hl-SpellCap

SpellCap Word that should start with a capital. spell
Combined with the highlighting used otherwise.

hl-SpellLocal

SpellLocal Word that is recognized by the spellchecker as one that is
used in another region. spell
Combined with the highlighting used otherwise.

hl-SpellRare

SpellRare Word that is recognized by the spellchecker as one that is
hardly ever used. spell
Combined with the highlighting used otherwise.

hl-StatusLine

StatusLine Status line of current window.

hl-StatusLineNC

StatusLineNC Status lines of not-current windows.

hl-StatusLineTerm

StatusLineTerm Status line of terminal window.

hl-StatusLineTermNC

StatusLineTermNC
Status line of non-current terminal windows.

hl-TabLine

TabLine Tabpages line, not active tabpage label.
Combined with hl-TabLineFill.

hl-TabLineFill

TabLineFill Tabpages line, where there are no labels.

hl-TabLineSel

TabLineSel Tabpages line, active tabpage label.
Combined with hl-TabLineFill.

hl-Title

Title Titles for output from ":set all", ":autocmd" etc.

hl-Visual

Visual Visual mode selection.

hl-VisualNOS

VisualNOS Visual mode selection when vim is "Not Owning the Selection".

hl-Whitespace

Whitespace "nbsp", "space", "tab", "multispace", "lead" and "trail"
in 'listchars'.

hl-WildMenu

WildMenu Current match in 'wildmenu' completion.

hl-WinBar

WinBar Window bar of current window.

hl-WinBarNC

WinBarNC Window bar of not-current windows.

hl-User1 hl-User1..9 hl-User9

The 'statusline' syntax allows the use of 9 different highlights in the
statusline and ruler (via 'rulerformat'). The names are User1 to User9.

For the GUI you can use the following groups to set the colors for the menu,
scrollbars and tooltips. They don't have defaults. This doesn't work for the
Win32 GUI. Only three highlight arguments have any effect here: font, guibg,
and guifg.

hl-Menu

Menu Current font, background and foreground colors of the menus.
Also used for the toolbar.
Applicable highlight arguments: font, guibg, guifg.

hl-Scrollbar

Scrollbar Current background and foreground of the main window's
scrollbars.
Applicable highlight arguments: guibg, guifg.

hl-Tooltip

Tooltip Current font, background and foreground of the tooltips.
Applicable highlight arguments: font, guibg, guifg.

14. Linking groups :hi-link :highlight-link E412 E413

When you want to use the same highlighting for several syntax groups, you can do this more easily by linking the groups into one common highlight group, and give the color attributes only for that group.

To set a link:

```
:hi[ghlight][!] [default] link {from-group} {to-group}
```

To remove a link:

```
:hi[ghlight][!] [default] link {from-group} NONE
```

Notes: E414

- If the {from-group} and/or {to-group} doesn't exist, it is created. You don't get an error message for a non-existing group.
- As soon as you use a ":highlight" command for a linked group, the link is removed.
- If there are already highlight settings for the {from-group}, the link is not made, unless the '!' is given. For a ":highlight link" command in a sourced file, you don't get an error message. This can be used to skip links for groups that already have settings.

:hi-default :highlight-default

The [default] argument is used for setting the default highlighting for a group. If highlighting has already been specified for the group the command will be ignored. Also when there is an existing link.

Using [default] is especially useful to overrule the highlighting of a specific syntax file. For example, the C syntax file contains:

```
:highlight default link cComment Comment
```

If you like Question highlighting for C comments, put this in your vimrc file:

```
:highlight link cComment Question
```

Without the "default" in the C syntax file, the highlighting would be overruled when the syntax file is loaded.

To have a link survive :highlight clear, which is useful if you have highlighting for a specific filetype and you want to keep it when selecting another color scheme, put a command like this in the "after/syntax/{filetype}.vim" file:

```
highlight! default link cComment Question
```

15. Cleaning up :syn-clear E391

If you want to clear the syntax stuff for the current buffer, you can use this command:

```
:syntax clear
```

This command should be used when you want to switch off syntax highlighting, or when you want to switch to using another syntax. It's normally not needed in a syntax file itself, because syntax is cleared by the autocommands that load the syntax file.
The command also deletes the "b:current_syntax" variable, since no syntax is loaded after this command.

To clean up specific syntax groups for the current buffer:

```
:syntax clear {group-name} ...
```

This removes all patterns and keywords for {group-name}.

To clean up specific syntax group lists for the current buffer:

```
:syntax clear @{grouplist-name} ...
```

This sets {grouplist-name}'s contents to an empty list.

:syntax-off :syn-off

If you want to disable syntax highlighting for all buffers, you need to remove the autocommands that load the syntax files:

```
:syntax off
```

What this command actually does, is executing the command

```
:source $VIMRUNTIME/syntax/nosyntax.vim
```

See the "nosyntax.vim" file for details. Note that for this to work $VIMRUNTIME must be valid. See $VIMRUNTIME.

:syntax-reset :syn-reset

If you have changed the colors and messed them up, use this command to get the defaults back:

```
:syntax reset
```

It is a bit of a wrong name, since it does not reset any syntax items, it only affects the highlighting.

Note that the syntax colors that you set in your vimrc file will also be reset back to their Vim default.
Note that if you are using a color scheme, the colors defined by the color scheme for syntax highlighting will be lost.

Note that when a color scheme is used, there might be some confusion whether your defined colors are to be used or the colors from the scheme. This depends on the color scheme file. See :colorscheme.

16. Highlighting tags

tag-highlight

If you want to highlight all the tags in your file, you can use the following mappings.

```
        <F11>   -- Generate tags.vim file, and highlight tags.
        <F12>   -- Just highlight tags based on existing tags.vim file.
```

```
:map <F11>  :sp tags<CR>:%s/^\([^       :]*:\)\=\([^    ]*\).*/syntax keyword Tag \2/<CR>:wq! tags.vim<CR>/^<CR><F12>
:map <F12>  :so tags.vim<CR>
```

WARNING: The longer the tags file, the slower this will be, and the more memory Vim will consume.

Only highlighting typedefs, unions and structs can be done too. For this you must use Universal Ctags (https://ctags.io) or Exuberant ctags.

Put these lines in your Makefile:

```
# Make a highlight file for types.  Requires Universal/Exuberant ctags and awk
types: types.vim
types.vim: *.[ch]
        ctags --c-kinds=gstu -o- *.[ch] |\
                awk 'BEGIN{printf("syntax keyword Type\t")}\
                        {printf("%s ", $$1)}END{print ""}' > $@
```

And put these lines in your vimrc:

```
" load the types.vim highlighting file, if it exists
autocmd BufRead,BufNewFile *.[ch] let fname = expand('<afile>:p:h') .. '/types.vim'
autocmd BufRead,BufNewFile *.[ch] if filereadable(fname)
autocmd BufRead,BufNewFile *.[ch]   exe 'so ' .. fname
autocmd BufRead,BufNewFile *.[ch] endif
```

17. Color xterms

xterm-color color-xterm

colortest.vim

To test your color setup, a file has been included in the Vim distribution.
To use it, execute this command:

```
:runtime syntax/colortest.vim
```

Nvim uses 256-color and true-color terminal capabilities wherever possible.

18. When syntax is slow

:synti :syntime

This is aimed at authors of a syntax file.

If your syntax causes redrawing to be slow, here are a few hints on making it faster. To see slowness switch on some features that usually interfere, such as 'relativenumber' and folding.

To find out what patterns are consuming the most time, get an overview with this sequence:

```
:syntime on
[ redraw the text at least once with CTRL-L ]
:syntime report
```

This will display a list of syntax patterns that were used, sorted by the time it took to match them against the text.

:synti[me] on — Start measuring syntax times. This will add some overhead to compute the time spent on syntax pattern matching.

:synti[me] off — Stop measuring syntax times.

:synti[me] clear — Set all the counters to zero, restart measuring.

:synti[me] report — Show the syntax items used since ":syntime on" in the current window. Use a wider display to see more of the output.

The list is sorted by total time. The columns are:

TOTAL	Total time in seconds spent on matching this pattern.
COUNT	Number of times the pattern was used.
MATCH	Number of times the pattern actually matched
SLOWEST	The longest time for one try.
AVERAGE	The average time for one try.
NAME	Name of the syntax item. Note that this is not unique.
PATTERN	The pattern being used.

Pattern matching gets slow when it has to try many alternatives. Try to include as much literal text as possible to reduce the number of ways a pattern does NOT match.

When using the "\@<=" and "\@<!" items, add a maximum size to avoid trying at all positions in the current and previous line. For example, if the item is literal text specify the size of that text (in bytes):

"<\@<=span"	Matches "span" in "<span". This tries matching with "<" in many places.
"<\@1<=span"	Matches the same, but only tries one byte before "span".

Editing with windows in multiple tabpages. tabpage

The commands which have been added to use multiple tabpages are explained here. Additionally, there are explanations for commands that work differently when used in combination with more than one tabpage.

1. Introduction tabpage-intro

A tabpage holds one or more windows. You can easily switch between tab pages, so that you have several collections of windows to work on different things.

Usually you will see a list of labels at the top of the Vim window, one for each tabpage. With the mouse you can click on the label to jump to that tab page. There are other ways to move between tabpages, see below.

Most commands work only in the current tabpage. That includes the CTRL-W commands, :windo, :all and :ball (when not using the :tab modifier).
The commands that are aware of other tabpages than the current one are mentioned below.

Tabpages are also a nice way to edit a buffer temporarily without changing the current window layout. Open a new tabpage, do whatever you want to do and close the tabpage.

tab-ID tabid tabpageid
Each tabpage has an unique identifier which will not change, even after rearranging tabs. nvim_get_current_tabpage() returns the tab-id and nvim_list_tabpages() lists tab-ids in the order they're displayed.

tabpage-number

Tabpages are numbered from left to right starting at 1: this tab "number" is returned by tabpagenr(). Unlike the tab-id, the tab number may change when tabs are opened, closed, or rearranged. nvim_tabpage_get_number() converts a tab-id to a tab number.

2. Commands tabpage-commands

OPENING A NEW TABPAGE:

When starting Vim "vim -p filename ..." opens each file argument in a separate tabpage (up to 'tabpagemax'). See -p

A double click with the mouse in the non-GUI tabpages line opens a new, empty tabpage. It is placed left of the position of the click. The first click may select another tabpage first, causing an extra screen update.

This also works in a few GUI versions, esp. Win32. But only when clicking right of the labels.

In the GUI tabpages line you can use the right mouse button to open menu. tabline-menu.

For the related autocommands see tabnew-autocmd.

:[count]tabe[dit] :tabe :tabedit :tabnew
:[count]tabnew
Open a new tabpage with an empty window, after the current tabpage. If [count] is given the new tabpage appears after the tabpage [count] otherwise the new tabpage will appear after the current one.

```
:tabnew     " opens tabpage after the current one
:.tabnew    " as above
:+tabnew    " opens tabpage after the next tabpage
            " note: it is one further than :tabnew
:-tabnew    " opens tabpage before the current
:0tabnew    " opens tabpage before the first one
:$tabnew    " opens tabpage after the last one
```

:[count]tabe[dit] [++opt] [+cmd] {file}
:[count]tabnew [++opt] [+cmd] {file}
Open a new tabpage and edit {file}, like with :edit.
For [count] see :tabnew above.

:[count]tabf[ind] [++opt] [+cmd] {file} :tabf :tabfind
Open a new tabpage and edit {file} in 'path', like with :find. For [count] see :tabnew above.

:[count]tab {cmd} :tab

Execute {cmd} and when it opens a new window open a new tab page instead. Doesn't work for :diffsplit, :diffpatch, :execute and :normal.
If [count] is given the new tabpage appears after the tab page [count] otherwise the new tabpage will appear after the current one.
Examples:

```
:tab split          " opens current buffer in new tabpage
:tab help gt    " opens tabpage with help for "gt"
:.tab help gt   " as above
:+tab help          " opens tabpage with help after the next
                " tabpage
:-tab help          " opens tabpage with help before the
                " current one
:0tab help          " opens tabpage with help before the
                " first one
:$tab help          " opens tabpage with help after the last
                " one
```

CTRL-W gf Open a new tabpage and edit the file name under the cursor.
           See CTRL-W_gf.

CTRL-W gF Open a new tabpage and edit the file name under the cursor
           and jump to the line number following the file name.
           See CTRL-W_gF.

CLOSING A TABPAGE:

Closing the last window of a tabpage closes the tabpage too, unless there is
only one tabpage.

Using the mouse: If the tabpage line is displayed you can click in the "X" at
the top right to close the current tabpage. A custom 'tabline' may show
something else.

:tabc :tabclose

:tabc[lose][!] Close current tabpage.
           This command fails when:

- There is only one tabpage on the screen. E784

- When 'hidden' is not set, [!] is not used, a buffer has

  changes, and there is no other window on this buffer.
  Changes to the buffer are not written and won't get lost, so
  this is a "safe" command.

```
:tabclose   " close the current tabpage
```

:{count}tabc[lose][!]
:tabc[lose][!] {count}
           Close tabpage {count}. Fails in the same way as :tabclose
           above.

```
:-tabclose          " close the previous tabpage
:+tabclose          " close the next tabpage
:1tabclose          " close the first tabpage
:$tabclose          " close the last tabpage
:tabclose -2    " close the 2nd previous tabpage
:tabclose +         " close the next tabpage
:tabclose 3         " close the third tabpage
:tabclose $         " close the last tabpage
:tabclose #     " close the last accessed tabpage
```

When a tabpage is closed the next tabpage will become the current one. This
behaviour can be customized using the 'tabclose' option.

:tabo :tabonly

:tabo[nly][!]   Close all other tabpages.
           When the 'hidden' option is set, all buffers in closed windows
           become hidden.
           When 'hidden' is not set, and the 'autowrite' option is set,
           modified buffers are written. Otherwise, windows that have
           buffers that are modified are not removed, unless the [!] is
           given, then they become hidden. But modified buffers are
           never abandoned, so changes cannot get lost.

```
:tabonly " close all tabpages except the current one
```

:tabo[nly][!] {count}
           Close all tabpages except {count} one.

```
:.tabonly           " as above
:-tabonly           " close all tabpages except the previous
                " one
:+tabonly           " close all tabpages except the next one
:1tabonly           " close all tabpages except the first one
:$tabonly           " close all tabpages except the last one
:tabonly -          " close all tabpages except the previous
                " one
:tabonly +2     " close all tabpages except the two next
                " one
:tabonly 1          " close all tabpages except the first one
:tabonly $          " close all tabpages except the last one
:tabonly #          " close all tabpages except the last
                " accessed one
```

SWITCHING TO ANOTHER TABPAGE:

Using the mouse: If the tabpage line is displayed you can click in a tabpage label to switch to that tabpage. Click where there is no label to go to the next tabpage. 'tabline'

:tabn[ext] :tabn :tabnext gt
<C-PageDown> CTRL-<PageDown> <C-PageDown>
gt i_CTRL-<PageDown> i_<C-PageDown>
Go to the next tabpage. Wraps around from the last to the first one.

:{count}tabn[ext]
:tabn[ext] {count}
Go to tabpage {count}. The first tabpage has number one.

```
:-tabnext       " go to the previous tabpage
:+tabnext       " go to the next tabpage
:+2tabnext      " go to the two next tabpage
:1tabnext       " go to the first tabpage
:$tabnext       " go to the last tabpage
:tabnext $      " as above
:tabnext #      " go to the last accessed tabpage
:tabnext -      " go to the previous tabpage
:tabnext -1     " as above
:tabnext +      " go to the next tabpage
:tabnext +1     " as above
```

{count}<C-PageDown>
{count}gt Go to tabpage {count}. The first tabpage has number one.

:tabp[revious] :tabp :tabprevious gT :tabN
:tabN[ext] :tabNext CTRL-<PageUp>
<C-PageUp> <C-PageUp> i_CTRL-<PageUp> i_<C-PageUp>
gT Go to the previous tabpage. Wraps around from the first one to the last one.

:tabp[revious] {count}
:tabN[ext] {count}
{count}<C-PageUp>
{count}gT Go {count} tabpages back. Wraps around from the first one to the last one. Note that the use of {count} is different from :tabnext, where it is used as the tabpage number.

:tabr[ewind] :tabfir :tabfirst :tabr :tabrewind
:tabfir[st] Go to the first tabpage.

:tabl :tablast
:tabl[ast] Go to the last tabpage.

<C-Tab> CTRL-<Tab> <C-Tab>
g<Tab> g<Tab> CTRL-W_g<Tab>
CTRL-W g<Tab> Go to the last accessed tabpage.

Other commands:

:tabs

:tabs List the tabpages and the windows they contain.
Shows a ">" for the current window.
Shows a "+" for modified buffers.
For example:

```
Tabpage 1
+ tabpage.txt
ex_docmd.c
Tabpage 2
>   main.c
```

REORDERING TABPAGES:

:tabm[ove] [N] :tabm :tabmove
:[N]tabm[ove]
Move the current tabpage to after tabpage N. Use zero to make the current tabpage the first one. N is counted before the move, thus if the second tabpage is the current one, :tabmove 1 and :tabmove 2 have no effect.
Without N the tabpage is made the last one.

```
:.tabmove       " do nothing
:-tabmove       " move the tabpage to the left
:+tabmove       " move the tabpage to the right
:0tabmove       " move the tabpage to the first
:tabmove 0      " as above
:tabmove        " move the tabpage to the last
:$tabmove       " as above
:tabmove $      " as above
:tabmove #      " move the tabpage after the last accessed
            " tabpage
```

```
:tabm[ove] +[N]
:tabm[ove] -[N]
                Move the current tabpage N places to the right (with +) or to
                the left (with -).
```

```
:tabmove -      " move the tabpage to the left
:tabmove -1     " as above
:tabmove +      " move the tabpage to the right
:tabmove +1     " as above
```

Note that although it is possible to move a tabpage behind the N-th one by using :Ntabmove. And move it by N places by using :+Ntabmove. For clarification what +N means in this context see [range].

LOOPING OVER TABPAGES:

:tabd :tabdo

```
:[range]tabd[o] {cmd}
                Execute {cmd} in each tabpage or, if [range] is given, only
                in tabpages which tabpage number is in the [range]. It works
                like doing this:
```

```
:tabfirst
:{cmd}
:tabnext
:{cmd}
etc.
```

```
                This only operates in the current window of each tabpage.
                When an error is detected on one tabpage, further tabpages
                will not be visited.
                The last tabpage (or where an error occurred) becomes the
                current tabpage.
                {cmd} can contain '|' to concatenate several commands.
                {cmd} must not open or close tabpages or reorder them.
                Also see :windo, :argdo, :bufdo, :cdo, :ldo, :cfdo
                and :lfdo.
```

3. Other items

tabpage-other

tabline-menu

The GUI tabpages line has a popup menu. It is accessed with a right click. The entries are:

Close	Close the tabpage under the mouse pointer. The current one if there is no label under the mouse pointer.
New Tab	Open a tabpage, editing an empty buffer. It appears to the left of the mouse pointer.
Open Tab...	Like "New Tab" and additionally use a file selector to select a file to edit.

Diff mode works per tabpage. You can see the diffs between several files within one tabpage. Other tabpages can show differences between other files.

Variables local to a tabpage start with "t:". tabpage-variable

Currently there is only one option local to a tabpage: 'cmdheight'.

tabnew-autocmd

The TabLeave and TabEnter autocommand events can be used to do something when switching from one tabpage to another. The exact order depends on what you are doing. When creating a new tabpage this works as if you create a new window on the same buffer and then edit another buffer. Thus ":tabnew" triggers:

WinLeave	leave current window
TabLeave	leave current tabpage
WinEnter	enter window in new tabpage
TabEnter	enter new tabpage
BufLeave	leave current buffer
BufEnter	enter new empty buffer

When switching to another tabpage the order is:

- BufLeave
- WinLeave
- TabLeave
- WinEnter

TabEnter
BufEnter

When entering a new tabpage (:tabnew), TabNew is triggered before TabEnter and after WinEnter.

4. Setting 'tabline' setting-tabline

The 'tabline' option specifies what the line with tabpages labels looks like. It is only used when there is no GUI tab line.

You can use the 'showtabline' option to specify when you want the line with tabpage labels to appear: never, when there is more than one tabpage or always.

The highlighting of the tabpages line is set with the groups TabLine TabLineSel and TabLineFill. hl-TabLine hl-TabLineSel hl-TabLineFill

A "+" will be shown for a tabpage that has a modified window. The number of windows in a tabpage is also shown. Thus "3+" means three windows and one of them has a modified buffer.

The 'tabline' option allows you to define your preferred way to tabpages labels. This isn't easy, thus an example will be given here.

For basics see the 'statusline' option. The same items can be used in the 'tabline' option. Additionally, the tabpagebuflist(), tabpagenr() and tabpagewinnr() functions are useful.

Since the number of tab labels will vary, you need to use an expression for the whole option. Something like:

```
:set tabline=%!MyTabLine()
```

Then define the MyTabLine() function to list all the tabpages labels. A convenient method is to split it in two parts: First go over all the tab pages and define labels for them. Then get the label for each tabpage.

```
function MyTabLine()
  let s = ''
  for i in range(tabpagenr('$'))
    " select the highlighting
    if i + 1 == tabpagenr()
      let s ..= '%#TabLineSel#'
    else
      let s ..= '%#TabLine#'
    endif

    " set the tabpage number (for mouse clicks)
    let s ..= '%' .. (i + 1) .. 'T'

    " the label is made by MyTabLabel()
    let s ..= ' %{MyTabLabel(' .. (i + 1) .. ')} '
  endfor

  " after the last tabpage fill with TabLineFill and reset tabpage nr
  let s ..= '%#TabLineFill#%T'

  " right-align the label to close the current tabpage
  if tabpagenr('$') > 1
    let s ..= '%=%#TabLine#%999Xclose'
  endif

  return s
endfunction
```

Now the MyTabLabel() function is called for each tabpage to get its label.

```
function MyTabLabel(n)
  let buflist = tabpagebuflist(a:n)
  let winnr = tabpagewinnr(a:n)
  return bufname(buflist[winnr - 1])
endfunction
```

This is just a simplistic example that results in a tabpages line that resembles the default, but without adding a + for a modified buffer or truncating the names. You will want to reduce the width of labels in a clever way when there is not enough room. Check the 'columns' option for the space available.

5. Setting 'guitablabel' setting-guitablabel

When the GUI tabpages line is displayed, 'guitablabel' can be used to specify the label to display for each tabpage. Unlike 'tabline', which specifies the whole tabpages line at once, 'guitablabel' is used for each label separately.

'guitabtooltip' is very similar and is used for the tooltip of the same label. This only appears when the mouse pointer hovers over the label, thus it usually is longer. Only supported on some systems though.

See the 'statusline' option for the format of the value.

The "%N" item can be used for the current tabpage number. The v:lnum
variable is also set to this number when the option is evaluated.
The items that use a file name refer to the current window of the tabpage.

Note that syntax highlighting is not used for the option. The %T and %X
items are also ignored.

A simple example that puts the tabpage number and the buffer name in the
label:

```
:set guitablabel=%N\ %f
```

An example that resembles the default 'guitablabel': Show the number of
windows in the tabpage and a '+' if there is a modified buffer:

```
function GuiTabLabel()
  let label = ''
  let bufnrlist = tabpagebuflist(v:lnum)

  " Add '+' if one of the buffers in the tabpage is modified
  for bufnr in bufnrlist
    if getbufvar(bufnr, "&modified")
      let label = '+'
      break
    endif
  endfor

  " Append the number of windows in the tabpage if more than one
  let wincount = tabpagewinnr(v:lnum, '$')
  if wincount > 1
    let label ..= wincount
  endif
  if label != ''
    let label ..= ' '
  endif

  " Append the buffer name
  return label .. bufname(bufnrlist[tabpagewinnr(v:lnum) - 1])
endfunction

set guitablabel=%{GuiTabLabel()}
```

Note that the function must be defined before setting the option, otherwise
you get an error message for the function not being known.

If you want to fall back to the default label, return an empty string.

If you want to show something specific for a tabpage, you might want to use a
tabpage local variable. t:var

Tags and special searches

See section 29.1 of the user manual for an introduction.

1. Jump to a tag

tag-commands

tag tags

A tag is an identifier that appears in a "tags" file. It is a sort of label that can be jumped to. For example: In C programs each function name can be used as a tag. The "tags" file has to be generated by a program like ctags, before the tag commands can be used.

With the ":tag" command the cursor will be positioned on the tag. With the CTRL-] command, the keyword on which the cursor is standing is used as the tag. If the cursor is not on a keyword, the first keyword to the right of the cursor is used.

The ":tag" command works very well for C programs. If you see a call to a function and wonder what that function does, position the cursor inside of the function name and hit CTRL-]. This will bring you to the function definition. An easy way back is with the CTRL-T command. Also read about the tag stack below.

:ta :tag E426 E429

:[count]ta[g][!] {name}
Jump to the definition of {name}, using the information in the tags file(s). Put {name} in the tag stack. See tag-! for [!].
{name} can be a regexp pattern, see tag-regexp.
When there are several matching tags for {name}, jump to the [count] one. When [count] is omitted the first one is jumped to. See tag-matchlist for jumping to other matching tags.

g<LeftMouse> g<LeftMouse> CTRL-]

CTRL-]
Jump to the definition of the keyword under the cursor. Same as ":tag {name}", where {name} is the keyword under or after cursor.
When there are several matching tags for {name}, jump to the [count] one. When no [count] is given the first one is jumped to. See tag-matchlist for jumping to other matching tags.

v CTRL-]

{Visual}CTRL-]
Same as ":tag {name}", where {name} is the text that is highlighted.

telnet-CTRL-]

CTRL-] is the default telnet escape key. When you type CTRL-] to jump to a tag, you will get the telnet prompt instead. Most versions of telnet allow changing or disabling the default escape key. See the telnet man page. You can telnet -E {Hostname} to disable the escape character, or
telnet -e {EscapeCharacter} {Hostname} to specify another escape character.
If possible, try to use "ssh" instead of "telnet" to avoid this problem.

tag-priority

When there are multiple matches for a tag, this priority is used:

1. "FSC" A full matching static tag for the current file.
2. "F C" A full matching global tag for the current file.
3. "F " A full matching global tag for another file.
4. "FS " A full matching static tag for another file.
5. " SC" An ignore-case matching static tag for the current file.
6. " C" An ignore-case matching global tag for the current file.
7. " " An ignore-case matching global tag for another file.
8. " S " An ignore-case matching static tag for another file.

Note that when the current file changes, the priority list is mostly not changed, to avoid confusion when using ":tnext". It is changed when using ":tag {name}".

The ignore-case matches are not found for a ":tag" command when:

- 'tagcase' is "followic" and the 'ignorecase' option is off
- 'tagcase' is "followscs" and the 'ignorecase' option is off and the

'smartcase' option is off or the pattern contains an upper case character.
- 'tagcase' is "match"

- 'tagcase' is "smart" and the pattern contains an upper case character.

The ignore-case matches are found when:
- a pattern is used (starting with a "/")

- for ":tselect"

- when 'tagcase' is "followic" and 'ignorecase' is on

- when 'tagcase' is "followscs" and 'ignorecase' is on or the 'smartcase'

option is on and the pattern does not contain an upper case character
- when 'tagcase' is "ignore"

- when 'tagcase' is "smart" and the pattern does not contain an upper case

character

Note that using ignore-case tag searching disables binary searching in the tags file, which causes a slowdown. This can be avoided by fold-case sorting the tag file. See the 'tagbsearch' option for an explanation.

2. Tag stack

tag-stack tagstack E425

On the tag stack is remembered which tags you jumped to, and from where. Tags are only pushed onto the stack when the 'tagstack' option is set.

g<RightMouse>

g<RightMouse>

<C-RightMouse>

<C-RightMouse> CTRL-T

CTRL-T Jump to [count] older entry in the tag stack (default 1).

:po :pop E555 E556

:[count]po[p][!] Jump to [count] older entry in tag stack (default 1). See tag-! for [!].

:[count]ta[g][!] Jump to [count] newer entry in tag stack (default 1). See tag-! for [!].

:tags

:tags Show the contents of the tag stack. The active entry is marked with a '>'.

The output of ":tags" looks like this:

```
   # TO tag      FROM line  in file/text
   1  1 main             1  harddisk2:text/vim/test
 > 2  2 FuncA           58  i = FuncA(10);
   3  1 FuncC          357  harddisk2:text/vim/src/amiga.c
```

This list shows the tags that you jumped to and the cursor position before that jump. The older tags are at the top, the newer at the bottom.

The '>' points to the active entry. This is the tag that will be used by the next ":tag" command. The CTRL-T and ":pop" command will use the position above the active entry.

Below the "TO" is the number of the current match in the match list. Note that this doesn't change when using ":pop" or ":tag".

The line number and file name are remembered to be able to get back to where you were before the tag command. The line number will be correct, also when deleting/inserting lines, unless this was done by another program (e.g. another instance of Vim).

For the current file, the "file/text" column shows the text at the position. An indent is removed and a long line is truncated to fit in the window.

You can jump to previously used tags with several commands. Some examples:

```
        ":pop" or CTRL-T        to position before previous tag
        {count}CTRL-T           to position before {count} older tag
        ":tag"                  to newer tag
        ":0tag"                 to last used tag
```

The most obvious way to use this is while browsing through the call graph of a program. Consider the following call graph:

```
        main  --->  FuncA  --->  FuncC
              --->  FuncB
```

(Explanation: main calls FuncA and FuncB; FuncA calls FuncC).
You can get from main to FuncA by using CTRL-] on the call to FuncA. Then you can CTRL-] to get to FuncC. If you now want to go back to main you can use CTRL-T twice. Then you can CTRL-] to FuncB.

If you issue a ":ta {name}" or CTRL-] command, this tag is inserted at the current position in the stack. If the stack was full (it can hold up to 20 entries), the oldest entry is deleted and the older entries shift one position up (their index number is decremented by one). If the last used entry was not at the bottom, the entries below the last used one are deleted. This means that an old branch in the call graph is lost. After the commands explained above the tag stack will look like this:

```
   # TO tag     FROM line  in file/text
   1  1 main            1  harddisk2:text/vim/test
   2  1 FuncB          59  harddisk2:text/vim/src/main.c
```

The gettagstack() function returns the tag stack of a specified window. The settagstack() function modifies the tag stack of a window.

tagstack-examples

Write to the tag stack just like :tag but with a user-defined jumper#jump_to_tag function:

```
" Store where we're jumping from before we jump.
let tag = expand('<cword>')
let pos = [bufnr()] + getcurpos()[1:]
let item = {'bufnr': pos[0], 'from': pos, 'tagname': tag}
if jumper#jump_to_tag(tag)
        " Jump was successful, write previous location to tag stack.
        let winid = win_getid()
        let stack = gettagstack(winid)
        let stack['items'] = [item]
        call settagstack(winid, stack, 't')
endif
```

Set current index of the tag stack to 4:

```
call settagstack(1005, {'curidx' : 4})
```

Push a new item onto the tag stack:

```
let pos = [bufnr('myfile.txt'), 10, 1, 0]
let newtag = [{'tagname' : 'mytag', 'from' : pos}]
call settagstack(2, {'items' : newtag}, 'a')
```

E73

When you try to use the tag stack while it doesn't contain anything you will get an error message.

3. Tag match list

tag-matchlist E427 E428

When there are several matching tags, these commands can be used to jump between them. Note that these commands don't change the tag stack, they keep the same entry.

:ts :tselect

:ts[elect][!] [name] List the tags that match [name], using the information in the tags file(s).
When [name] is not given, the last tag name from the tag stack is used.
See tag-! for [!].
With a '>' in the first column is indicated which is the current position in the list (if there is one).
[name] can be a regexp pattern, see tag-regexp.
See tag-priority for the priorities used in the listing.
Example output:

```
  # pri kind tag               file
  1 F   f    mch_delay         os_amiga.c
               mch_delay(msec, ignoreinput)
> 2 F   f    mch_delay         os_msdos.c
               mch_delay(msec, ignoreinput)
  3 F   f    mch_delay         os_unix.c
               mch_delay(msec, ignoreinput)
Type number and <Enter> (empty cancels):
```

See tag-priority for the "pri" column. Note that this depends on the current file, thus using ":tselect xxx" can produce different results.
The "kind" column gives the kind of tag, if this was included in the tags file.
The "info" column shows information that could be found in the tags file. It depends on the program that produced the tags file.
When the list is long, you may get the more-prompt. If you already see the tag you want to use, you can type 'q' and enter the number.

:sts :stselect

:sts[elect][!] [name] Does ":tselect[!] [name]" and splits the window for the selected tag.

g]

g] Like CTRL-], but use ":tselect" instead of ":tag".

v_g]

{Visual}g] Same as "g]", but use the highlighted text as the identifier.

:tj :tjump

:tj[ump][!] [name] Like ":tselect", but jump to the tag directly when there is only one match.

:stj :stjump

:stj[ump][!] [name] Does ":tjump[!] [name]" and splits the window for the selected tag.

g_CTRL-]

g CTRL-] Like CTRL-], but use ":tjump" instead of ":tag".

v_g_CTRL-]

{Visual}g CTRL-] Same as "g CTRL-]", but use the highlighted text as the identifier.

:tn :tnext

:[count]tn[ext][!] Jump to [count] next matching tag (default 1). See tag-! for [!].

]t

]t Mapped to :tnext. default-mappings

:tp :tprevious

:[count]tp[revious][!] Jump to [count] previous matching tag (default 1). See tag-! for [!].

[t

[t Mapped to :tprevious. default-mappings

:tN :tNext

:[count]tN[ext][!] Same as ":tprevious".

:tr :trewind

:[count]tr[ewind][!] Jump to first matching tag. If [count] is given, jump to [count]th matching tag. See tag-! for [!].

[T

[T Mapped to :trewind. default-mappings

:tf :tfirst

:[count]tf[irst][!] Same as ":trewind".

:tl :tlast

:tl[ast][!] Jump to last matching tag. See tag-! for [!].

]T

]T Mapped to :tlast. default-mappings

:lt :ltag

:lt[ag][!] [name] Jump to tag [name] and add the matching tags to a new location list for the current window. [name] can be a regexp pattern, see tag-regexp. When [name] is not given, the last tag name from the tag stack is used. The search pattern to locate the tag line is prefixed with "\V" to escape all the special characters (very nomagic). The location list showing the matching tags is independent of the tag stack. See tag-! for [!].

When there is no other message, Vim shows which matching tag has been jumped to, and the number of matching tags:

```
tag 1 of 3 or more
```

The " or more" is used to indicate that Vim didn't try all the tags files yet. When using ":tnext" a few times, or with ":tlast", more matches may be found.

When you didn't see this message because of some other message, or you just want to know where you are, this command will show it again (and jump to the same tag as last time):

```
:0tn
```

tag-skip-file

When a matching tag is found for which the file doesn't exist, this match is skipped and the next matching tag is used. Vim reports this, to notify you of missing files. When the end of the list of matches has been reached, an error message is given.

tag-preview

The tag match list can also be used in the preview window. The commands are the same as above, with a "p" prepended.

:pts :ptselect

:pts[elect][!] [name] Does ":tselect[!] [name]" and shows the new tag in a

"Preview" window. See :ptag for more info.

:ptj :ptjump
:ptj[ump][!] [name] Does ":tjump[!] [name]" and shows the new tag in a "Preview" window. See :ptag for more info.

:ptn :ptnext
:[count]ptn[ext][!] ":tnext" in the preview window. See :ptag.

]_CTRL-T

]_CTRL-T Mapped to :ptnext. default-mappings

:ptp :ptprevious
:[count]ptp[revious][!] ":tprevious" in the preview window. See :ptag.

[_CTRL-T

[_CTRL-T Mapped to :ptprevious. default-mappings

:ptN :ptNext
:[count]ptN[ext][!] Same as ":ptprevious".

:ptr :ptrewind
:[count]ptr[ewind][!] ":trewind" in the preview window. See :ptag.

:ptf :ptfirst
:[count]ptf[irst][!] Same as ":ptrewind".

:ptl :ptlast
:ptl[ast][!] ":tlast" in the preview window. See :ptag.

4. Tags details tag-details

static-tag

A static tag is a tag that is defined for a specific file. In a C program this could be a static function.

In Vi jumping to a tag sets the current search pattern. This means that the "n" command after jumping to a tag does not search for the same pattern that it did before jumping to the tag. Vim does not do this as we consider it to be a bug. If you really want the old Vi behavior, set the 't' flag in 'cpoptions'.

tag-binary-search

Vim uses binary searching in the tags file to find the desired tag quickly. But this only works if the tags file was sorted on ASCII byte value. Therefore, if no match was found, another try is done with a linear search. If you only want the linear search, reset the 'tagbsearch' option. Or better: Sort the tags file!

Note that the binary searching is disabled when not looking for a tag with a specific name. This happens when ignoring case and when a regular expression is used that doesn't start with a fixed string. Tag searching can be a lot slower then. The former can be avoided by case-fold sorting the tags file. See 'tagbsearch' for details.

tag-regexp

The ":tag" and ":tselect" commands accept a regular expression argument. See pattern for the special characters that can be used.
When the argument starts with '/', it is used as a pattern. If the argument does not start with '/', it is taken literally, as a full tag name.
Examples:

```
:tag main
```

jumps to the tag "main" that has the highest priority.

```
:tag /^get
```

jumps to the tag that starts with "get" and has the highest priority.

```
:tag /norm
```

lists all the tags that contain "norm", including "id_norm".
When the argument both exists literally, and match when used as a regexp, a literal match has a higher priority. For example, ":tag /open" matches "open" before "open_file" and "file_open".
When using a pattern case is ignored. If you want to match case use "\C" in the pattern.

tag-!

If the tag is in the current file this will always work. Otherwise the performed actions depend on whether the current file was changed, whether a ! is added to the command and on the 'autowrite' and 'winfixbuf' options:

tag in file	winfixbuf	autowrite		
current file changed ! option option action				
yes	x	x off	x	goto tag
no	no	x off	x	read other file, goto tag
no	yes	yes off	x	abandon current file, read other file, goto tag
no	yes	no off	on	write current file, read other file, goto tag
no	yes	no off	off	fail
yes	x	yes x	x	goto tag
no	no	no on	x	fail
no	yes	no on	x	fail
no	yes	no on	on	fail
no	yes	no on	off	fail

- If the tag is in the current file, the command will always work.

- If the tag is in another file and the current file was not changed, the

  other file will be made the current file and read into the buffer.
- If the tag is in another file, the current file was changed and a ! is

  added to the command, the changes to the current file are lost, the other
  file will be made the current file and read into the buffer.
- If the tag is in another file, the current file was changed and the

  'autowrite' option is on, the current file will be written, the other
  file will be made the current file and read into the buffer.
- If the tag is in another file, the current file was changed and the

  'autowrite' option is off, the command will fail. If you want to save
  the changes, use the ":w" command and then use ":tag" without an argument.
  This works because the tag is put on the stack anyway. If you want to lose
  the changes you can use the ":tag!" command.
- If the tag is in another file and the window includes 'winfixbuf', the

  command will fail. If the tag is in the same file then it may succeed.

tag-security

Note that Vim forbids some commands, for security reasons. This works like
using the 'secure' option for exrc/vimrc files in the current directory. See
trojan-horse and sandbox.
When the {tagaddress} changes a buffer, you will get a warning message:
        "WARNING: tag command changed a buffer!!!"
In a future version changing the buffer will be impossible. All this for
security reasons: Somebody might hide a nasty command in the tags file, which
would otherwise go unnoticed. Example:

```
:$d|/tag-function-name/
```

In Vi the ":tag" command sets the last search pattern when the tag is searched
for. In Vim this is not done, the previous search pattern is still
remembered, unless the 't' flag is present in 'cpoptions'.

tags-option

The 'tags' option is a list of file names. Each of these files is searched
for the tag. This can be used to use a different tags file than the default
file "tags". It can also be used to access a common tags file.

The next file in the list is not used when:
- A matching static tag for the current buffer has been found.

- A matching global tag has been found.

  This also depends on whether case is ignored. Case is ignored when:
- 'tagcase' is "followic" and 'ignorecase' is set

- 'tagcase' is "ignore"

- 'tagcase' is "smart" and the pattern only contains lower case

  characters.
- 'tagcase' is "followscs" and 'smartcase' is set and the pattern only

contains lower case characters.
If case is not ignored, and the tags file only has a match without matching case, the next tags file is searched for a match with matching case. If no tag with matching case is found, the first match without matching case is used. If case is ignored, and a matching global tag with or without matching case is found, this one is used, no further tags files are searched.

When a tag file name starts with "./", the '.' is replaced with the path of the current file. This makes it possible to use a tags file in the directory where the current file is (no matter what the current directory is). The idea of using "./" is that you can define which tag file is searched first: In the current directory ("tags,./tags") or in the directory of the current file ("./tags,tags").

For example:

```
:set tags=./tags,tags,/home/user/commontags
```

In this example the tag will first be searched for in the file "tags" in the directory where the current file is. Next the "tags" file in the current directory. If it is not found there, then the file "/home/user/commontags" will be searched for the tag.

This can be switched off by including the 'd' flag in 'cpoptions', to make it Vi compatible. "./tags" will then be the tags file in the current directory, instead of the tags file in the directory where the current file is.

Instead of the comma a space may be used. Then a backslash is required for the space to be included in the string option:

```
:set tags=tags\ /home/user/commontags
```

To include a space in a file name use three backslashes. To include a comma in a file name use two backslashes. For example, use:

```
:set tags=tag\\\ file,/home/user/common\\,tags
```

for the files "tag file" and "/home/user/common,tags". The 'tags' option will have the value "tag\ file,/home/user/common\,tags".

If the 'tagrelative' option is on (which is the default) and using a tag file in another directory, file names in that tag file are relative to the directory where the tag file is.

5. Tags file format

tags-file-format E431

ctags

A tags file can be created with an external command, for example "ctags". It will contain a tag for each function. Some versions of "ctags" will also make a tag for each "#defined" macro, typedefs, enums, etc.

Some programs that generate tags files:

ctags	As found on most Unix systems. Only supports C. Only does the basic work.
universal ctags	A maintained version of ctags based on exuberant ctags. See https://ctags.io.

Exuberant_ctags

exuberant ctags	Works for C, C++, Java, Fortran, Eiffel and others. See https://ctags.sourceforge.net. No new version since 2009.
:helptags	For Vim's help files
ptags.py	For Python, in Python. Found in your Python source directory at Tools/scripts/ptags.py.
ptags	For Perl, in Perl. It can be found at https://metacpan.org/pod/Vim::Tag

The lines in the tags file must have one of these two formats:

1. {tagname} {TAB} {tagfile} {TAB} {tagaddress}

2. {tagname} {TAB} {tagfile} {TAB} {tagaddress} {term} {field} ..

Previously an old format was supported, see tag-old-static.

The first format is a normal tag, which is completely compatible with Vi. It is the only format produced by traditional ctags implementations. This is often used for functions that are global, also referenced in other files.

The lines in the tags file can end in <NL> or <CR><NL>. On the Macintosh <CR> also works. The <CR> and <NL> characters can never appear inside a line.

The second format is new. It includes additional information in optional fields at the end of each line. It is backwards compatible with Vi. It is only supported by new versions of ctags (such as Universal ctags or Exuberant ctags).

{tagname}	The identifier. Normally the name of a function, but it can
		be any identifier. It cannot contain a <Tab>.
{TAB}		One <Tab> character. **Note:** previous versions allowed any
		white space here. This has been abandoned to allow spaces in
		{tagfile}.
{tagfile}	The file that contains the definition of {tagname}. It can
		have an absolute or relative path. It may contain environment
		variables and wildcards (although the use of wildcards is
		doubtful). It cannot contain a <Tab>.
{tagaddress}	The Ex command that positions the cursor on the tag. It can
		be any Ex command, although restrictions apply (see
		tag-security). Posix only allows line numbers and search
		commands, which are mostly used.
{term}		;" The two characters semicolon and double quote. This is
		interpreted by Vi as the start of a comment, which makes the
		following be ignored. This is for backwards compatibility
		with Vi, it ignores the following fields. Example:

```
APP     file    /^static int APP;$/;"   v
```

		When {tagaddress} is not a line number or search pattern, then
		{term} must be |;". Here the bar ends the command (excluding
		the bar) and ;" is used to have Vi ignore the rest of the
		line. Example:

```
APP     file.c  call cursor(3, 4)|;"    v
```

{field} ..	A list of optional fields. Each field has the form:

			<Tab>{fieldname}:{value}

		The {fieldname} identifies the field, and can only contain
		alphabetical characters [a-zA-Z].
		The {value} is any string, but cannot contain a <Tab>.
		These characters are special:
			"\t" stands for a <Tab>
			"\r" stands for a <CR>
			"\n" stands for a <NL>
			"\\" stands for a single '\' character

		There is one field that doesn't have a ':'. This is the kind
		of the tag. It is handled like it was preceded with "kind:".
		In the above example, this was "kind:v" (typically variable).
		See the documentation of ctags for the kinds it produces, with
		ctags you can use ctags --list-kinds .

		The only other field currently recognized by Vim is "file:"
		(with an empty value). It is used for a static tag.

The first lines in the tags file can contain lines that start with
	!_TAG_
These are sorted to the first lines, only rare tags that start with "!" can
sort to before them. Vim recognizes two items. The first one is the line
that indicates if the file was sorted. When this line is found, Vim uses
binary searching for the tags file:
!_TAG_FILE_SORTED<Tab>1<Tab>{anything}

A tag file may be case-fold sorted to avoid a linear search when case is
ignored. (Case is ignored when 'ignorecase' is set and 'tagcase' is
"followic", or when 'tagcase' is "ignore".) See 'tagbsearch' for details.
The value '2' should be used then:
!_TAG_FILE_SORTED<Tab>2<Tab>{anything}

The other tag that Vim recognizes is the encoding of the tags file:
!_TAG_FILE_ENCODING<Tab>utf-8<Tab>{anything}
Here "utf-8" is the encoding used for the tags. Vim will then convert the tag
being searched for from 'encoding' to the encoding of the tags file. And when
listing tags the reverse happens. When the conversion fails the unconverted
tag is used.

tag-search

The command can be any Ex command, but often it is a search command.
Examples:
**tag1	file1	/^main(argc, argv)/**
**tag2	file2	108**

The command is always executed with 'magic' not set. The only special
characters in a search pattern are "^" (begin-of-line) and "$" (<EOL>).
See pattern. Note that you must put a backslash before each backslash in
the search text. This is for backwards compatibility with Vi.

E434 E435

If the command is a normal search command (it starts and ends with "/" or
"?"), some special handling is done:
- Searching starts on line 1 of the file.

  The direction of the search is forward for "/", backward for "?".
  Note that 'wrapscan' does not matter, the whole file is always searched.
- If the search fails, another try is done ignoring case. If that fails too,

  a search is done for:

```
"^tagname[ \t]*("
```

(the tag with '^' prepended and "[\t]*(" appended). When using function names, this will find the function name when it is in column 0. This will help when the arguments to the function have changed since the tags file was made. If this search also fails another search is done with:

```
"^[#a-zA-Z_].*\<tagname[ \t]*("
```

This means: A line starting with '#' or an identifier and containing the tag followed by white space and a '('. This will find macro names and function names with a type prepended.

tag-old-static

Until March 2019 (patch 8.1.1092) an outdated format was supported:
{tagfile}:{tagname} {TAB} {tagfile} {TAB} {tagaddress}

This format is for a static tag only. It is obsolete now, replaced by the second format. It is only supported by Elvis 1.x, older Vim versions and a few versions of ctags. A static tag is often used for functions that are local, only referenced in the file {tagfile}. Note that for the static tag, the two occurrences of {tagfile} must be exactly the same. Also see tags-option below, for how static tags are used.

The support was removed, since when you can update to the new Vim version you should also be able to update ctags to one that supports the second format.

6. Include file searches include-search definition-search

E387 E388 E389

These commands look for a string in the current file and in all encountered included files (recursively). This can be used to find the definition of a variable, function or macro. If you only want to search in the current buffer, use the commands listed at pattern-searches.

When a line is encountered that includes another file, that file is searched before continuing in the current buffer. Files included by included files are also searched. When an include file could not be found it is silently ignored. Use the :checkpath command to discover which files could not be found, possibly your 'path' option is not set up correctly. **Note:** the included file is searched, not a buffer that may be editing that file. Only for the current file the lines in the buffer are used.

The string can be any keyword or a defined macro. For the keyword any match will be found. For defined macros only lines that match with the 'define' option will be found. The default is "^#\s*define", which is for C programs. For other languages you probably want to change this. See 'define' for an example for C++. The string cannot contain an end-of-line, only matches within a line are found.

When a match is found for a defined macro, the displaying of lines continues with the next line when a line ends in a backslash.

The commands that start with "[" start searching from the start of the current file. The commands that start with "]" start at the current cursor position.

The 'include' option is used to define a line that includes another file. The default is "\^#\s*include", which is for C programs. **Note:** Vim does not recognize C syntax, if the 'include' option matches a line inside "#ifdef/#endif" or inside a comment, it is searched anyway. The 'isfname' option is used to recognize the file name that comes after the matched pattern.

The 'path' option is used to find the directory for the include files that do not have an absolute path.

The 'comments' option is used for the commands that display a single line or jump to a line. It defines patterns that may start a comment. Those lines are ignored for the search, unless [!] is used. One exception: When the line matches the pattern "^# *define" it is not considered to be a comment.

If you want to list matches, and then select one to jump to, you could use a mapping to do that for you. Here is an example:

```
:map <F4> [I:let nr = input("Which one: ")<Bar>exe "normal " .. nr .. "[\t"<CR>
```

[i

[i Display the first line that contains the keyword under the cursor. The search starts at the beginning of the file. Lines that look like a comment are ignored (see 'comments' option). If a count is given, the count'th matching line is displayed, and comment lines are not ignored.

]i

]i like "[i", but start at the current cursor position.

:is :isearch

:[range]is[earch][!] [count] [/]pattern[/]
Like "[i" and "]i", but search in [range] lines

(default: whole file).
See :search-args for [/] and [!].

[I

[I Display all lines that contain the keyword under the cursor. Filenames and line numbers are displayed for the found lines. The search starts at the beginning of the file.

]I

]I like "[I", but start at the current cursor position.

:il :ilist

:[range]il[ist][!] [/]pattern[/]
Like "[I" and "]I", but search in [range] lines (default: whole file).
See :search-args for [/] and [!].

[_CTRL-I

[CTRL-I Jump to the first line that contains the keyword under the cursor. The search starts at the beginning of the file. Lines that look like a comment are ignored (see 'comments' option). If a count is given, the count'th matching line is jumped to, and comment lines are not ignored.

]_CTRL-I

] CTRL-I like "[CTRL-I", but start at the current cursor position.

:ij :ijump

:[range]ij[ump][!] [count] [/]pattern[/]
Like "[CTRL-I" and "] CTRL-I", but search in [range] lines (default: whole file).
See :search-args for [/] and [!].

CTRL-W CTRL-I CTRL-W_CTRL-I CTRL-W_i
CTRL-W i Open a new window, with the cursor on the first line that contains the keyword under the cursor. The search starts at the beginning of the file. Lines that look like a comment line are ignored (see 'comments' option). If a count is given, the count'th matching line is jumped to, and comment lines are not ignored.

:isp :isplit

:[range]isp[lit][!] [count] [/]pattern[/]
Like "CTRL-W i" and "CTRL-W i", but search in [range] lines (default: whole file).
See :search-args for [/] and [!].

[d

[d Display the first macro definition that contains the macro under the cursor. The search starts from the beginning of the file. If a count is given, the count'th matching line is displayed.

[d-default

Jumps to the previous diagnostic in the current buffer by default. vim.diagnostic.jump() default-mappings

]d

]d like "[d", but start at the current cursor position.

]d-default

Jumps to the next diagnostic in the current buffer by default. vim.diagnostic.jump() default-mappings

:ds :dsearch

:[range]ds[earch][!] [count] [/]string[/]
Like "[d" and "]d", but search in [range] lines (default: whole file).
See :search-args for [/] and [!].

[D

[D Display all macro definitions that contain the macro under the cursor. Filenames and line numbers are displayed for the found lines. The search starts from the beginning of the file.

[D-default

Jumps to the first diagnostic in the current buffer by default. vim.diagnostic.jump() default-mappings

]D

]D like "[D", but start at the current cursor position.

]D-default

Jumps to the last diagnostic in the current buffer by
default. vim.diagnostic.jump() default-mappings

:dli :dlist

:[range]dli[st][!] [/]string[/]
Like [D and]D, but search in [range] lines
(default: whole file).
See :search-args for [/] and [!].
Note: :dl works like :delete with the "l" flag,
not :dlist.

[_CTRL-D

[CTRL-D Jump to the first macro definition that contains the
keyword under the cursor. The search starts from
the beginning of the file. If a count is given, the
count'th matching line is jumped to.

]_CTRL-D

] CTRL-D like "[CTRL-D", but start at the current cursor
position.

:dj :djump

:[range]dj[ump][!] [count] [/]string[/]
Like "[CTRL-D" and "] CTRL-D", but search in
[range] lines (default: whole file).
See :search-args for [/] and [!].

CTRL-W CTRL-D CTRL-W_CTRL-D CTRL-W_d
CTRL-W d Open a new window, with the cursor on the first
macro definition line that contains the keyword
under the cursor. The search starts from the
beginning of the file. If a count is given, the
count'th matching line is jumped to.

CTRL-W_d-default

Mapped to vim.diagnostic.open_float() by default.
default-mappings

:dsp :dsplit

:[range]dsp[lit][!] [count] [/]string[/]
Like "CTRL-W d", but search in [range] lines
(default: whole file).
See :search-args for [/] and [!].

:checkp :checkpath

:checkp[ath] List all the included files that could not be found.

:checkp[ath]! List all the included files.

:search-args

Common arguments for the commands above:
[!] When included, find matches in lines that are recognized as comments.
When excluded, a match is ignored when the line is recognized as a
comment (according to 'comments'), or the match is in a C comment
(after "//" or inside /* */). Note that a match may be missed if a
line is recognized as a comment, but the comment ends halfway the
line. And if the line is a comment, but it is not recognized
(according to 'comments') a match may be found in it anyway. Example:

```
/* comment
   foobar */
```

A match for "foobar" is found, because this line is not recognized as
a comment (even though syntax highlighting does recognize it).
Note: Since a macro definition mostly doesn't look like a comment, the
[!] makes no difference for ":dlist", ":dsearch" and ":djump".
[/] A pattern can be surrounded by "/". Without "/" only whole words are
matched, using the pattern "\<pattern\>". Only after the second "/" a
next command can be appended with "|". Example:

```
:isearch /string/ | echo "the last one"
```

For a ":djump", ":dsplit", ":dlist" and ":dsearch" command the pattern
is used as a literal string, not as a search pattern.

7. Using 'tagfunc' tag-function

It is possible to provide Vim with a function which will generate a list of
tags used for commands like :tag, :tselect, Normal mode tag commands like
CTRL-] and for the taglist() function.

The function used for generating the taglist is specified by setting the
'tagfunc' option. The function will be called with three arguments:
pattern The tag identifier or pattern used during the tag search.
flags String containing flags to control the function behavior.
info Dict containing the following entries:
buf_ffname Full filename which can be used for priority.
user_data Custom data String, if stored in the tag
stack previously by tagfunc.

Note that "a:" needs to be prepended to the argument name when using it.

Currently up to three flags may be passed to the tag function:

'c'	The function was invoked by a normal command being processed (mnemonic: the tag function may use the context around the cursor to perform a better job of generating the tag list.)
'i'	In Insert mode, the user was completing a tag (with i_CTRL-X_CTRL-] or 'complete' contains "`t`" or "`]`").
'r'	The first argument to tagfunc should be interpreted as a pattern (see tag-regexp), such as when using:

```
:tag /pat
```

It is also given when completing in insert mode.
If this flag is not present, the argument is usually taken literally as the full tag name.

Note that when 'tagfunc' is set, the priority of the tags described in tag-priority does not apply. Instead, the priority is exactly as the ordering of the elements in the list returned by the function.

E987

The function should return a List of Dict entries. Each Dict must at least include the following entries and each value must be a string:

name	Name of the tag.
filename	Name of the file where the tag is defined. It is either relative to the current directory or a full path.
cmd	Ex command used to locate the tag in the file. This can be a line number, a search pattern or any other Ex command, like in a tags file (see tags-file-format). A search pattern must start and end with "/" or "?". An invalid value results in E987.

Note that the format is similar to that of taglist(), which makes it possible to use its output to generate the result.
The following fields are optional:

kind	Type of the tag.
user_data	String of custom data stored in the tag stack which can be used to disambiguate tags between operations.

If the function returns v:null instead of a List, a standard tag lookup will be performed instead.

It is not allowed to change the tagstack from inside 'tagfunc'. E986
It is not allowed to close a window or change window from inside 'tagfunc'.
E1299

The following is a hypothetical example of a function used for 'tagfunc'. It uses the output of taglist() to generate the result: a list of tags in the inverse order of file names.

```
vim.bo.tagfunc = function(pattern, flags, info)
  local result = vim.fn.taglist(pattern)
  table.sort(result, function(item1, item2)
    return item1.filename > item2.filename
  end)
  return result
end
```

Note: When executing taglist() the 'tagfunc' function won't be called recursively.

Terminal emulator terminal-emulator

Nvim embeds a VT220/xterm terminal emulator based on libvterm. The terminal is presented as a special 'buftype', asynchronously updated as data is received from the connected program.

Terminal buffers behave like normal buffers, except:
- With 'modifiable', lines can be edited but not deleted.
- 'scrollback' controls how many lines are kept.
- Output is followed ("tailed") if cursor is on the last line.
- 'modified' is the default. You can set 'nomodified' to avoid a warning when closing the terminal buffer.
- 'bufhidden' defaults to "hide".

Start terminal-start

There are several ways to create a terminal buffer:

- Run the :terminal command.
- Call nvim_open_term() or `jobstart(…, {'term': v:true})`.
- Edit a "term://" buffer. Examples:

```
:edit term://bash
:vsplit term://top
```

Note: To open a "term://" buffer from an autocmd, the autocmd-nested modifier is required.

```
autocmd VimEnter * ++nested split term://sh
```

(This is only mentioned for reference; use :terminal instead.)

When the terminal starts, the buffer contents are updated and the buffer is named in the form of `term://{cwd}//{pid}:{cmd}`. This naming scheme is used by :mksession to restore a terminal buffer (by restarting the `{cmd}`).

The terminal environment is initialized as in jobstart-env.

MS-Windows ConPTY terminal-ms-windows terminal-conpty

The system ConPTY (in "kernel32.dll") mangles some VT sequences, which can garble :terminal output. To use a newer ConPTY from the Windows Terminal project, put its "conpty.dll" (and "OpenConsole.exe") next to the Nvim executable. Nvim uses that "conpty.dll" if present, else falls back to the system ConPTY, and logs which one ($NVIM_LOG_FILE):

```
Loaded new ConPTY from conpty.dll
Loaded system ConPTY from kernel32.dll
```

Input terminal-input

To send input, enter Terminal-mode with i, I, a, A or :startinsert. In this mode all keys except <C-\> are sent to the underlying program. If <C-\> is pressed, the next key is sent unless it is <C-N> or <C-O>. Use <C-\><C-N> to return to normal mode. CTRL-_CTRL-N Use <C-\><C-O> to execute one normal mode command and then return to terminal mode. t_CTRL-_CTRL-O

Terminal-mode forces these local options:

- 'cursorlineopt' = number
- 'nocursorcolumn'
- 'scrolloff' = 0
- 'sidescrolloff' = 0

Terminal-mode has its own :tnoremap namespace for mappings, this can be used to automate any terminal interaction.

To map <Esc> to exit terminal-mode:

```
:tnoremap <Esc> <C-\><C-n>
```

To simulate i_CTRL-R in terminal-mode:

```
:tnoremap <expr> <C-R> '<C-\><C-N>"'.nr2char(getchar()).'pi'
```

To use ALT+{h,j,k,l} to navigate windows from any mode:

```
:tnoremap <A-h> <C-\><C-N><C-w>h
:tnoremap <A-j> <C-\><C-N><C-w>j
:tnoremap <A-k> <C-\><C-N><C-w>k
:tnoremap <A-l> <C-\><C-N><C-w>l
:inoremap <A-h> <C-\><C-N><C-w>h
:inoremap <A-j> <C-\><C-N><C-w>j
:inoremap <A-k> <C-\><C-N><C-w>k
:inoremap <A-l> <C-\><C-N><C-w>l
:nnoremap <A-h> <C-w>h
:nnoremap <A-j> <C-w>j
:nnoremap <A-k> <C-w>k
:nnoremap <A-l> <C-w>l
```

You can also create menus similar to terminal mode mappings, but you have to use :tlmenu instead of :tmenu.

Mouse input has the following behavior:

- If the program has enabled mouse events, the corresponding events will be forwarded to the program.

- If mouse events are disabled (the default), terminal focus will be lost and the event will be processed as in a normal buffer.
- If another window is clicked, terminal focus will be lost and nvim will jump to the clicked window
- If the mouse wheel is used while the mouse is positioned in another window, the terminal won't lose focus and the hovered window will be scrolled.

Configuration terminal-config

- Options: 'modified', 'scrollback'
- Events: TermOpen, TermEnter, TermLeave, TermClose
- Event groups: "nvim.terminal" default-autocmds
- Highlight groups: hl-TermCursor

Terminal sets local defaults for some options, which may differ from your global configuration.

- 'list' is disabled
- 'wrap' is disabled
- 'number' is disabled
- 'relativenumber' is disabled
- 'signcolumn' is set to "no"
- 'foldcolumn' is set to "0"

You can change the defaults with a TermOpen autocommand:

```
:autocmd TermOpen * setlocal list
```

By default, "[Process exited]" is displayed when the terminal process ends. To disable that, you can delete the "nvim.terminal" TermClose handler (this also deletes other default-autocmds TermClose functionality):

```
:autocmd! nvim.terminal TermClose
```

TERMINAL COLORS

The `{g,b}:terminal_color_x` variables control the terminal color palette, where x is the color index between 0 and 15 inclusive. The variables are read during TermOpen. The value must be a color name or hexadecimal string. Example:

```
let g:terminal_color_4 = '#ff0000'
let g:terminal_color_5 = 'green'
```

Only works for RGB UIs (see 'termguicolors'); for 256-color terminals the color index is just forwarded.

Editor highlighting (syntax-highlighting, highlight-groups, etc.) has higher precedence: it is applied after terminal colors are resolved.

EVENTS terminal-events

Applications running in a :terminal buffer can send requests, which Nvim exposes via the TermRequest event.

OSC 7: change working directory terminal-osc7

Shells can emit the "OSC 7" sequence to announce when the current directory (CWD) changed.

You can configure your shell init (e.g. ~/.bashrc) to emit OSC 7, or your terminal may attempt to do it for you.

To configure bash to emit OSC 7:

```
function print_osc7() {
  printf '\033]7;file://%s\033\\' "$PWD"
}
PROMPT_COMMAND='print_osc7'
```

Having ensured that your shell emits OSC 7, you can now handle it in Nvim. The following code runs :bcd whenever your shell CWD changes, so the :terminal buffer-local directory always follows the shell:

```
vim.api.nvim_create_autocmd('TermRequest', {
  desc = 'Handles OSC 7 dir change requests',
  callback = function(ev)
    local val, n = string.gsub(ev.data.sequence, '\027]7;file://[^/]*', '')
    if n > 0 then
      -- OSC 7: dir-change
      local dir = val
      if vim.fn.isdirectory(dir) == 0 then
        vim.notify('invalid dir: '..dir)
        return
      end
      vim.cmd.bcd(dir)
    end
  end
})
```

To try it out, select the above code and source it with `:'<,'>lua`, then run this command in a :terminal buffer:

```
printf "\033]7;file://./foo/bar\033\\"
```

OSC 52: write to system clipboard terminal-osc52

Applications in the :terminal buffer can write to the system clipboard by emitting an OSC 52 sequence. Example:

```
printf '\033]52;;%s\033\\' "$(echo -n 'Hello world' | base64)"
```

Nvim uses the configured clipboard provider to write to the system clipboard. Reading from the system clipboard with OSC 52 is not supported, as this would allow any arbitrary program in the :terminal to read the user's clipboard.

OSC 52 sequences sent from the :terminal buffer do not emit a TermRequest event. The event is handled directly by Nvim and is not forwarded to plugins.

OSC 133: shell integration terminal-osc133 shell-prompt

Shells can emit semantic escape sequences (OSC 133) to mark where each prompt starts and ends. The start of a prompt is marked by sequence `OSC 133 ; A ST`, and the end by `OSC 133 ; B ST`.

You can configure your shell init (e.g. ~/.bashrc) to emit OSC 133 sequences, or your terminal may attempt to do it for you (assuming your shell config doesn't interfere).

- fish: https://fishshell.com/docs/current/relnotes.html#improved-terminal-support
- kitty: https://sw.kovidgoyal.net/kitty/shell-integration/
- powershell: https://learn.microsoft.com/en-us/windows/terminal/tutorials/shell-integration#powershell-pwshexe
- vscode: https://code.visualstudio.com/docs/terminal/shell-integration

To configure bash to mark the start of each prompt, set $PROMPT_COMMAND:

```
# Prompt start:
PROMPT_COMMAND='printf "\033]133;A\007"'
```

terminal_]] terminal_[[The]] and [[motions jump to the next/previous prompts, if your shell emits OSC 133 as described above.

shell-prompt-signs

To annotate each terminal prompt with a sign, call nvim_buf_set_extmark() from a TermRequest handler:

```
vim.api.nvim_create_autocmd('TermOpen', {
  command = 'setlocal signcolumn=auto',
})
local ns = vim.api.nvim_create_namespace('my.terminal.prompt')
vim.api.nvim_create_autocmd('TermRequest', {
  callback = function(ev)
    if string.match(ev.data.sequence, '^\027]133;A') then
      local lnum = ev.data.cursor[1]
      vim.api.nvim_buf_set_extmark(ev.buf, ns, lnum - 1, 0, {
        sign_text = '▶',
        sign_hl_group = 'SpecialChar',
      })
    end
  end,
})
```

Status Variables terminal-status

Terminal buffers maintain some buffer-local variables and options. The values are initialized before TermOpen, so you can use them in a local 'statusline'. Example:

```
:autocmd TermOpen * setlocal statusline=%{b:term_title}
```

- b:term_title Terminal title (user-writable), typically displayed in the window title or tab title of a graphical terminal emulator. Terminal programs can set this by emitting an escape sequence.
- 'channel' Terminal PTY job-id. Can be used with chansend() to send input to the terminal.
- The TermClose event gives the terminal job exit code in the v:event "status" field. For example, this autocommand outputs the terminal's exit code to :messages:

  ```
  autocmd TermClose * echom 'Terminal exited with status '..v:event.status
  ```

  Use jobwait() to check if the terminal job has finished:

  ```
  let running = jobwait([&channel], 0)[0] == -1
  ```

Concepts terminal-concepts

TODO

:Termdebug plugin terminal-debug terminal-debugger

package-termdebug termdebug

The Terminal debugging plugin can be used to debug a program with gdb and view the source code in a Vim window. Since this is completely contained inside Vim this also works remotely over an ssh connection.

Starting

termdebug-starting

Load the plugin with this command:

```
packadd termdebug
```

When loading the plugin from the vimrc file, add the "!" attribute:

```
packadd! termdebug
```

:Termdebug

To start debugging use `:Termdebug` or `:TermdebugCommand` followed by the command name, for example:

```
:Termdebug vim
```

This opens two windows:

- gdb window: A terminal window in which "gdb vim" is executed. Here you can directly interact with gdb.
- program window: A terminal window for the executed program. When "run" is used in gdb the program I/O will happen in this window, so that it does not interfere with controlling gdb.

The current window is used to show the source code. When gdb pauses the source file location will be displayed, if possible. A sign is used to highlight the current position, using highlight group debugPC.

If the buffer in the current window is modified, another window will be opened to display the current gdb position.

Focus the terminal of the executed program to interact with it. This works the same as any command running in a terminal window.

When the debugger ends, typically by typing "quit" in the gdb window, the two opened windows are closed.

Only one debugger can be active at a time. :TermdebugCommand
If you want to give specific commands to the command being debugged, you can use the `:TermdebugCommand` command followed by the command name and additional parameters.

```
:TermdebugCommand vim --clean -c ':set nu'
```

Both the `:Termdebug` and `:TermdebugCommand` support an optional "!" bang argument to start the command right away, without pausing at the gdb window (and cursor will be in the debugged window). For example:

```
:TermdebugCommand! vim --clean
```

To attach gdb to an already running executable or use a core file, pass extra arguments. E.g.:

```
:Termdebug vim core
:Termdebug vim 98343
```

If no argument is given, you'll end up in a gdb window, in which you need to specify which command to run using e.g. the gdb `file` command.

Example session

termdebug-example

Start in the Vim "src" directory and build Vim:

```
% make
```

Start Vim:

```
% ./vim
```

Load the termdebug plugin and start debugging Vim:

```
:packadd termdebug
:Termdebug vim
```

You should now have three windows:
- source: where you started
- gdb: you can type gdb commands here
- program: the executed program will use this window

Put focus on the gdb window and type:

```
break ex_help
run
```

Vim will start running in the program window. Put focus there and type:

```
:help gui
```

Gdb will run into the ex_help breakpoint. The source window now shows the ex_cmds.c file. A red "1 " marker will appear in the signcolumn where the breakpoint was set. The line where the debugger stopped is highlighted. You can now step through the program. You will see the highlighting move as the debugger executes a line of source code.

Run :Over a few times until the for loop is highlighted. Put the cursor on the end of "eap->arg", then call ":Eval". You will see this displayed:

```
"eap->arg": 0x555555e68855 "gui"
```

This way you can inspect the value of local variables. You can also focus the gdb window and use a "print" command, e.g.:

```
print *eap
```

If mouse pointer movements are working, Vim will also show a balloon when the mouse rests on text that can be evaluated by gdb. You can also use the "K" mapping that will either use Nvim floating windows to show the results.

Now go back to the source window and put the cursor on the first line after the for loop, then type:

```
:Break
```

You will see a "1" marker appear, this indicates the new breakpoint. Now run ":Cont" command and the code until the breakpoint will be executed.

You can type more advanced commands in the gdb window. For example, type:

```
watch curbuf
```

Now run ":Cont" (or type "cont" in the gdb window). Execution will now continue until the value of "curbuf" changes, which is in do_ecmd(). To remove this watchpoint again type in the gdb window:

```
delete 3
```

You can see the stack by typing in the gdb window:

```
where
```

Move through the stack frames, e.g. with:

```
frame 3
```

The source window will show the code, at the point where the call was made to a deeper level.

Stepping through code

termdebug-stepping

Put focus on the gdb window to type commands there. Some common ones are:
- CTRL-C: interrupt the program
- next: execute the current line and stop at the next line
- step: execute the current line and stop at the next statement, entering functions
- until: execute until past the current cursor line or past a specified position or the current stack frame returns
- finish: execute until leaving the current function
- where: show the stack
- frame N: go to the Nth stack frame
- continue: continue execution

:Run :Arguments In the window showing the source code these commands can be used to control gdb:
- `:Run` [args] run the program with [args] or the previous arguments
- `:Arguments {args}` set arguments for the next `:Run`
- :Break set a breakpoint at the cursor position
- :Break `[{position}]` [thread `{nr}`] [if `{expr}`] set a breakpoint at the specified position if `{position}` is omitted, use the current file and line thread `{nr}` limits the breakpoint to one thread if `{expr}` sets a conditional breakpoint. Examples:

  ```
  :Break if argc == 1
  :Break 42 thread 3 if x > 10
  :Break main
  ```

- :Tbreak set a temporary breakpoint at the cursor position
- :Tbreak `[{position}]` [thread `{nr}`] [if `{expr}`] like `:Break`, but the breakpoint is deleted after it is hit once. Examples:

  ```
  :Tbreak if argc == 1
  :Tbreak 42 thread 3 if x > 10
  :Tbreak main
  ```

- :Clear delete the breakpoint at the cursor position
- :ToggleBreak set a breakpoint at the cursor position or delete all breakpoints at the cursor positoin
- :Step execute the gdb "step" command
- :Over execute the gdb "next" command (`:Next` is a Vim command)
- :Until execute the gdb "until" command
- :Finish execute the gdb "finish" command
- :Continue execute the gdb "continue" command
- :RunOrContinue execute the gdb "continue" command if program is running, otherwise run the program
- :Stop interrupt the program

If gdb stops at a source line and there is no window currently showing the source code, a new window will be created for the source code. This also happens if the buffer in the source code window has been modified and can't be abandoned.

Gdb gives each breakpoint a number. In Vim the number shows up in the sign column, with a red background. You can use these gdb commands:
- info break list breakpoints
- delete N delete breakpoint N You can also use the `:Clear` command if the cursor is in the line with the breakpoint, or use the "Clear breakpoint" right-click menu entry.

Inspecting variables

termdebug-variables :Evaluate
- `:Evaluate` evaluate the expression under the cursor
- K same (see termdebug_map_K to disable)
- `:Evaluate {expr}` evaluate `{expr}`
- `:'<,'>Evaluate` evaluate the Visually selected text

This is similar to using "print" in the gdb window. You can usually shorten `:Evaluate` to `:Ev`. The result is displayed in a floating window. You can move the cursor to this window by running `:Evaluate` (or K) again.

Navigating stack frames

termdebug-frames :Frame :Up :Down
- `:Frame` [frame] select frame [frame], which is a frame number, address, or function name (default: current frame)

- :Up [count] go up [count] frames (default: 1; the frame that called the current)
- + same (see termdebug_map_plus to disable)
- :Down [count] go down [count] frames (default: 1; the frame called by the current)
- - same (see termdebug_map_minus to disable)

Other commands

termdebug-commands

- :Gdb jump to the gdb window
- :Program jump to the window with the running program
- :Source jump to the window with the source code, create it if there isn't one
- :Asm jump to the window with the disassembly, create it if there isn't one
- :Var jump to the window with the local and argument variables, create it if there isn't one. This window updates whenever the program is stopped

Events

termdebug-events

Four autocommands can be used:

```
au User TermdebugStartPre  echomsg 'debugging starting'
au User TermdebugStartPost echomsg 'debugging started'
au User TermdebugStopPre   echomsg 'debugging stopping'
au User TermdebugStopPost  echomsg 'debugging stopped'
```

- TermdebugStartPre Before starting debugging. Not triggered if the debugger is already running or the debugger command cannot be executed.
- TermdebugStartPost After debugging has initialized. If a "!" bang is passed to :Termdebug or :TermdebugCommand the event is triggered before running the provided command in gdb.
- TermdebugStopPre Before debugging ends, when gdb is terminated, most likely after issuing a "quit" command in the gdb window.
- TermdebugStopPost After debugging has ended, gdb-related windows are closed, debug buffers wiped out and the state before the debugging was restored.

Customizing

termdebug-customizing g:termdebug_config In the past several global variables were used for configuration. These are deprecated and using the g:termdebug_config dictionary is preferred. When g:termdebug_config exists the other global variables will NOT be used. The recommended way is to start with an empty dictionary:

```
let g:termdebug_config = {}
```

Then you can add entries to the dictionary as mentioned below. The deprecated global variable names are mentioned for completeness. If you are switching over to using g:termdebug_config you can find the old variable name and take over the value, then delete the deprecated variable.

Prompt mode

termdebug-prompt

When on MS-Windows, gdb will run in a buffer with 'buftype' set to "prompt". This works slightly differently:

- The gdb window will be in Insert mode while typing commands. Go to Normal mode with <Esc>, then you can move around in the buffer, copy/paste, etc. Go back to editing the gdb command with any command that starts Insert mode, such as a or i.
- A separate :terminal window will be opened to run the debugged program in.

termdebug_use_prompt

Prompt mode can be used with:

```
let g:termdebug_config['use_prompt'] = 1
```

If there is no g:termdebug_config you can use:

```
let g:termdebug_use_prompt = 1
```

Mappings

The termdebug plugin enables a few default mappings. All those mappings are reset to their original values once the termdebug session concludes.

termdebug_map_K termdebug-mappings The K key is normally mapped to :Evaluate unless a buffer local (:map-local) mapping to K already exists. If you do not want this use:

```
let g:termdebug_config['map_K'] = 0
```

If there is no g:termdebug_config you can use:

```
let g:termdebug_map_K = 0
```

termdebug_map_minus

The - key is normally mapped to :Down unless a buffer local mapping to the - key already exists. If you do not want this use:

```
let g:termdebug_config['map_minus'] = 0
```

termdebug_map_plus

The + key is normally mapped to :Up unless a buffer local mapping to the + key already exists. If you do not want this use:

```
let g:termdebug_config['map_plus'] = 0
```

termdebug_disasm_window

If you want the Asm window shown by default, set the "disasm_window" flag to
1. The "disasm_window_height" entry can be used to set the window height:

```
let g:termdebug_config['disasm_window'] = 1
let g:termdebug_config['disasm_window_height'] = 15
```

If there is no g:termdebug_config you can use:

```
let g:termdebug_disasm_window = 15
```

Any value greater than 1 will set the Asm window height to that value. If the current window has enough horizontal space, it will be vertically split and the Asm window will be shown side by side with the source code window (and the height option won't be used).

termdebug_variables_window

If you want the Var window shown by default, set the "variables_window" flag to 1. The "variables_window_height" entry can be used to set the window height:

```
let g:termdebug_config['variables_window'] = 1
let g:termdebug_config['variables_window_height'] = 15
```

If there is no g:termdebug_config you can use:

```
let g:termdebug_variables_window = 15
```

Any value greater than 1 will set the Var window height to that value. If the current window has enough horizontal space, it will be vertically split and the Var window will be shown side by side with the source code window (and the height options won't be used).

Communication

termdebug-communication

There is another, hidden, buffer, which is used for Vim to communicate with gdb. The buffer name is "gdb communication". Do not delete this buffer, it will break the debugger.

Gdb has some weird behavior, the plugin does its best to work around that. For example, after typing "continue" in the gdb window a `CTRL-C` can be used to interrupt the running program. But after using the MI command "-exec-continue" pressing `CTRL-C` does not interrupt. Therefore you will see "continue" being used for the `:Continue` command, instead of using the communication channel.

GDB command

g:termdebugger

To change the name of the gdb command, set "debugger" entry in g:termdebug_config or the "g:termdebugger" variable before invoking `:Termdebug`:

```
let g:termdebug_config['command'] = "mygdb"
```

If there is no g:termdebug_config you can use:

```
let g:termdebugger = "mygdb"
```

If the command needs an argument use a List:

```
let g:termdebug_config['command'] = ['rr', 'replay', '--']
```

If there is no g:termdebug_config you can use:

```
let g:termdebugger = ['rr', 'replay', '--']
```

If you are a mouse person, you can also define a mapping using your right click to one of the terminal command like evaluate the variable under the cursor:

```
nnoremap <RightMouse> :Evaluate<CR>
```

or set/unset a breakpoint:

```
nnoremap <RightMouse> :Break<CR>
```

Several arguments will be added to make gdb work well for the debugger. If you want to modify them, add a function to filter the argument list:

```
let g:termdebug_config['command_filter'] = MyDebugFilter
```

If you do not want the arguments to be added, but you do need to set the "pty", use a function to add the necessary arguments:

```
let g:termdebug_config['command_add_args'] = MyAddArguments
```

The function will be called with the list of arguments so far, and a second argument that is the name of the pty.

gdb-version

Only debuggers fully compatible with gdb will work. Vim uses the GDB/MI interface. The "new-ui" command requires gdb version 7.12 or later. If you get this error:

```
Undefined command: "new-ui". Try "help".
```

Then your gdb is too old.

Colors

hl-debugPC hl-debugBreakpoint The color of the signs can be adjusted with these highlight groups:

- debugPC the current position
- debugBreakpoint a breakpoint

The defaults are, when 'background' is "light":

```
hi debugPC term=reverse ctermbg=lightblue guibg=lightblue
hi debugBreakpoint term=reverse ctermbg=red guibg=red
```

When 'background' is "dark":

```
hi debugPC term=reverse ctermbg=darkblue guibg=darkblue
hi debugBreakpoint term=reverse ctermbg=red guibg=red
```

Shortcuts

termdebug_shortcuts

You can define your own shortcuts (mappings) to control gdb, that can work in any window, using the TermDebugSendCommand() function. Example:

```
map ,w :call TermDebugSendCommand('where')<CR>
```

The argument is the gdb command.

Popup menu

termdebug_popup

By default the Termdebug plugin sets 'mousemodel' to "popup_setpos" and adds these entries to the popup menu:

- Set breakpoint `:Break`
- Clear breakpoint `:Clear`
- Evaluate `:Evaluate` If you don't want this then disable it with:

```
let g:termdebug_config['popup'] = 0
```

If there is no g:termdebug_config you can use:

```
let g:termdebug_popup = 0
```

Change default signs

termdebug_signs

Termdebug uses the hex number of the breakpoint ID in the signcolumn to represent breakpoints. If it is greater than "0xFF", then it will be displayed as "F+", because there are only two screen cells available for the sign. You may also use decimal breakpoint signs instead, in which case IDs greater than 99 will be displayed as "9+".

If you want to customize the breakpoint signs to show >> in the signcolumn:

```
let g:termdebug_config['sign'] = '>>'
```

You can also specify individual signs for the first several breakpoints:

```
let g:termdebug_config['signs'] = ['>1', '>2', '>3', '>4', '>5', '>6', '>7', '>8', '>9']
let g:termdebug_config['sign'] = '>>'
```

If you would like to use decimal (base 10) breakpoint signs:

```
let g:termdebug_config['sign_decimal'] = 1
```

If the variable g:termdebug_config does not yet exist, you can use:

```
let g:termdebug_config = {'sign': '>>'}
```

Likewise, to enable decimal signs:

```
let g:termdebug_config = {'sign_decimal': 1}
```

Vim window width

termdebug_wide

To change the width of the Vim window when debugging starts and use a vertical split:

```
let g:termdebug_config['wide'] = 163
```

If there is no g:termdebug_config you can use:

```
let g:termdebug_wide = 163
```

This will set 'columns' to 163 when `:Termdebug` is used. The value is restored when quitting the debugger.

If the wide value is set and 'columns' is already a greater value, then a vertical split will be used without modifying 'columns'.

Set the wide value to 1 to use a vertical split without ever changing 'columns'. This is useful when the terminal can't be resized by Vim.

Tips and ideas for using Vim

These are just a few that we thought would be helpful for many users.
You can find many more tips on the wiki. The URL can be found on
https://www.vim.org

Don't forget to browse the user manual, it also contains lots of useful tips
usr_toc.txt.

Editing C programs

C-editing

There are quite a few features in Vim to help you edit C program files. Here
is an overview with tags to jump to:

usr_29.txt	Moving through programs chapter in the user manual.
usr_30.txt	Editing programs chapter in the user manual.
C-indenting	Automatically set the indent of a line while typing text.
=	Re-indent a few lines.
format-comments	Format comments.
:checkpath	Show all recursively included files.
[i	Search for identifier under cursor in current and included files.
[_CTRL-I	Jump to match for "[i"
[I	List all lines in current and included files where identifier under the cursor matches.
[d	Search for define under cursor in current and included files.
CTRL-]	Jump to tag under cursor (e.g., definition of a function).
CTRL-T	Jump back to before a CTRL-] command.
:tselect	Select one tag out of a list of matching tags.
gd	Go to Declaration of local variable under cursor.
gD	Go to Declaration of global variable under cursor.
gf	Go to file name under the cursor.
%	Go to matching (), {}, [], /* */, #if, #else, #endif.
[/	Go to previous start of comment.
]/	Go to next end of comment.
[#	Go back to unclosed #if, #ifdef, or #else.
]#	Go forward to unclosed #else or #endif.
[(	Go back to unclosed '('
])	Go forward to unclosed ')'
[{	Go back to unclosed '{'
]}	Go forward to unclosed '}'
v_ab	Select "a block" from "[(" to "])", including braces
v_ib	Select "inner block" from "[(" to "])"
v_aB	Select "a block" from [{ to]}, including brackets
v_iB	Select "inner block" from [{ to]}

Finding where identifiers are used

ident-search

You probably already know that tags can be used to jump to the place where a
function or variable is defined. But sometimes you wish you could jump to all
the places where a function or variable is being used. This is possible in
two ways:

1. Using the :grep command. This should work on most Unix systems,
   but can be slow (it reads all files) and only searches in one directory.
2. Using ID utils. This is fast and works in multiple directories. It uses a
   database to store locations. You will need some additional programs for
   this to work. And you need to keep the database up to date.

Using the GNU id-tools:

What you need:

- The GNU id-tools installed (mkid is needed to create ID and lid is needed to
  use the macros).
- An identifier database file called "ID" in the current directory. You can
  create it with the shell command "mkid file1 file2 ..".

Put this in your init.vim:

```
map _u :call ID_search()<Bar>execute "/\\<" .. g:word .. "\\>"<CR>
map _n :n<Bar>execute "/\\<" .. g:word .. "\\>"<CR>

function! ID_search()
  let g:word = expand("<cword>")
  let x = system("lid --key=none " .. g:word)
  let x = substitute(x, "\n", " ", "g")
  execute "next " .. x
endfun
```

To use it, place the cursor on a word, type "_u" and vim will load the file that contains the word. Search for the next occurrence of the word in the same file with "n". Go to the next file with "_n".

This has been tested with id-utils-3.2 (which is the name of the id-tools archive file on your closest gnu-ftp-mirror).

[the idea for this comes from Andreas Kutschera]

Scrolling in Insert mode

scroll-insert

If you are in insert mode and you want to see something that is just off the screen, you can use CTRL-X CTRL-E and CTRL-X CTRL-Y to scroll the screen.
i_CTRL-X_CTRL-E

To make this easier, you could use these mappings:

```
:inoremap <C-E> <C-X><C-E>
:inoremap <C-Y> <C-X><C-Y>
```

You then lose the ability to copy text from the line above/below the cursor i_CTRL-E.

Also consider setting 'scrolloff' to a larger value, so that you can always see some context around the cursor. If 'scrolloff' is bigger than half the window height, the cursor will always be in the middle and the text is scrolled when the cursor is moved up/down.

Smooth scrolling

scroll-smooth

If you like the scrolling to go a bit smoother, you can use these mappings:

```
:map <C-U> <C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y><C-Y>
:map <C-D> <C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E><C-E>
```

Correcting common typing mistakes

type-mistakes

When there are a few words that you keep on typing in the wrong way, make abbreviations that correct them. For example:

```
:ab teh the
:ab fro for
```

Counting words, lines, etc.

count-items

To count how often any pattern occurs in the current buffer use the substitute command and add the 'n' flag to avoid the substitution. The reported number of substitutions is the number of items. Examples:

```
:%s/./&/gn              characters
:%s/\i\+/&/gn           words
:%s/^//n                lines
:%s/the/&/gn            "the" anywhere
:%s/\<the\>/&/gn        "the" as a word
```

You might want to reset 'hlsearch' or do ":nohlsearch".
Add the 'e' flag if you don't want an error when there are no matches.

An alternative is using v_g_CTRL-G in Visual mode.

If you want to find matches in multiple files use :vimgrep.

count-bytes

If you want to count bytes, you can use this:

```
        Visually select the characters (block is also possible)
        Use "y" to yank the characters
        Use the strlen() function:
```

```
:echo strlen(@")
```

A line break is counted for one byte.

Restoring the cursor position

restore-position

Sometimes you want to write a mapping that makes a change somewhere in the file and restores the cursor position, without scrolling the text. For example, to change the date mark in a file:

```
:map <F2> msHmtgg/Last [cC]hange:\s*/e+1<CR>"_D"=strftime("%Y %b %d")<CR>p'tzt`s
```

Breaking up saving the position:

```
        ms      store cursor position in the 's' mark
```

```
        H       go to the first line in the window
        mt      store this position in the 't' mark
```

Breaking up restoring the position:

```
        't      go to the line previously at the top of the window
        zt      scroll to move this line to the top of the window
        `s      jump to the original position of the cursor
```

For something more advanced see winsaveview() and winrestview().

Renaming files

rename-files

Say I have a directory with the following files in them (directory picked at random :-):

```
buffer.c
charset.c
digraph.c

...
```

and I want to rename *.c *.bla. I'd do it like this:

```
$ vim
:r !ls *.c
:%s/\(.*\).c/mv & \1.bla
:w !sh
:q!
```

Change a name in multiple files

change-name

Example for using a script file to change a name in several files:

Create a file "subs.vim" containing substitute commands and a :update command:

```
:%s/Jones/Smith/g
:%s/Allen/Peter/g
:update
```

Execute Vim on all files you want to change, and source the script for each argument:

```
vim *.let
argdo source subs.vim
```

See :argdo.

Speeding up external commands

speed-up

In some situations, execution of an external command can be very slow. This can also slow down wildcard expansion on Unix. Here are a few suggestions to increase the speed.

If your .cshrc (or other file, depending on the shell used) is very long, you should separate it into a section for interactive use and a section for non-interactive use (often called secondary shells). When you execute a command from Vim like ":!ls", you do not need the interactive things (for example, setting the prompt). Put the stuff that is not needed after these lines:

```
if ($?prompt == 0) then
        exit 0
endif
```

Another way is to include the "-f" flag in the 'shell' option, e.g.:

```
:set shell=csh\ -f
```

(the backslash is needed to include the space in the option).
This will make csh completely skip the use of the .cshrc file. This may cause some things to stop working though.

Useful mappings

useful-mappings

Here are a few mappings that some people like to use.

map-backtick

```
:map ' `
```

Make the single quote work like a backtick. Puts the cursor on the column of a mark, instead of going to the first non-blank character in the line.

emacs-keys

For Emacs-style editing on the command-line:

```
" start of line
:cnoremap <C-A>         <Home>
" back one character
:cnoremap <C-B>         <Left>
" delete character under cursor
:cnoremap <C-D>         <Del>
" end of line
:cnoremap <C-E>         <End>
" forward one character
:cnoremap <C-F>         <Right>
" recall newer command-line
:cnoremap <C-N>         <Down>
" recall previous (older) command-line
:cnoremap <C-P>         <Up>
" back one word
:cnoremap <Esc><C-B>    <S-Left>
" forward one word
:cnoremap <Esc><C-F>    <S-Right>
```

format-bullet-list

This mapping will format any bullet list. It requires that there is an empty
line above and below each list entry. The expression commands are used to
be able to give comments to the parts of the mapping.

```
:let m =     ":map _f  :set ai<CR>"   " need 'autoindent' set
:let m ..= "{O<Esc>"                  " add empty line above item
:let m ..= "}{()^W"                   " move to text after bullet
:let m ..= "i     <CR>     <Esc>"     " add space for indent
:let m ..= "gq}"                      " format text after the bullet
:let m ..= "{dd"                      " remove the empty line
:let m ..= "5lDJ"                     " put text after bullet
:execute m                            |" define the mapping
```

(<> notation <>. Note that this is all typed literally. ^W is "^" "W", not

CTRL-W.)

Note that the last comment starts with |", because the ":execute" command
doesn't accept a comment directly.

You also need to set 'textwidth' to a non-zero value, e.g.,

```
:set tw=70
```

A mapping that does about the same, but takes the indent for the list from the
first line (**Note:** this mapping is a single long line with a lot of spaces):

```
:map _f :set ai<CR>}{a                                                  <Esc>WWmmkD`mi<CR><Esc>kkddpJgq}'mJO<Esc>j
```

collapse

These two mappings reduce a sequence of empty (;b) or blank (;n) lines into a
single line

```
:map ;b   GoZ<Esc>:g/^$/.,/./-j<CR>Gdd
:map ;n   GoZ<Esc>:g/^[ <Tab>]*$/.,/[^ <Tab>]/-j<CR>Gdd
```

Compressing the help files

gzip-helpfile

For those of you who are really short on disk space, you can compress the help
files and still be able to view them with Vim. This makes accessing the help
files a bit slower and requires the "gzip" program.

(1) Compress all the help files: "gzip doc/*.txt".

(2) Edit "doc/tags" and change the ".txt" to ".txt.gz":

```
:%s=\(\t.*\.txt\)\t=\1.gz\t=
```

(3) Add this line to your vimrc:

```
set helpfile={dirname}/help.txt.gz
```

Where {dirname} is the directory where the help files are. The gzip plugin
will take care of decompressing the files.
You must make sure that $VIMRUNTIME is set to where the other Vim files are,
when they are not in the same location as the compressed "doc" directory. See
$VIMRUNTIME.

Hex editing

hex-editing using-xxd

See section 23.3 of the user manual.

If one has a particular extension that one uses for binary files (such as exe, bin, etc), you may find it helpful to automate the process with the following bit of autocmds for your init.vim. Change that "*.bin" to whatever comma-separated list of extension(s) you find yourself wanting to edit:

```
" vim -b : edit binary using xxd-format!
augroup Binary
  autocmd!
  autocmd BufReadPre  *.bin set binary
  autocmd BufReadPost *.bin
    \ if &binary
    \ |   execute "silent %!xxd -c 32"
    \ |   set filetype=xxd
    \ |   redraw
    \ | endif
  autocmd BufWritePre *.bin
    \ if &binary
    \ |   let s:view = winsaveview()
    \ |   execute "silent %!xxd -r -c 32"
    \ | endif
  autocmd BufWritePost *.bin
    \ if &binary
    \ |   execute "silent %!xxd -c 32"
    \ |   set nomodified
    \ |   call winrestview(s:view)
    \ |   redraw
    \ | endif
augroup END
```

Using <> notation in autocommands

autocmd-<>

The <> notation is not recognized in the argument of an :autocmd. To avoid having to use special characters, you could use a self-destroying mapping to get the <> notation and then call the mapping from the autocmd. Example:

map-self-destroy

```
" This is for automatically adding the name of the file to the menu list.
" It uses a self-destroying mapping!
" 1. use a line in the buffer to convert the 'dots' in the file name to \.
" 2. store that in register '"'
" 3. add that name to the Buffers menu list
" WARNING: this does have some side effects, like overwriting the
" current register contents and removing any mapping for the "i" command.
"
autocmd BufNewFile,BufReadPre * nmap i :nunmap i<CR>O<C-R>%<Esc>:.g/\./s/\./\\./g<CR>0"9y$u:menu Buffers.<C-R>9 :buffer <C-R>%<C-V><CR>
<CR>
autocmd BufNewFile,BufReadPre * normal i
```

Another method, perhaps better, is to use the ":execute" command. In the string you can use the <> notation by preceding it with a backslash. Don't forget to double the number of existing backslashes and put a backslash before '"'.

```
autocmd BufNewFile,BufReadPre * exe "normal O\<C-R>%\<Esc>:.g/\\./s/\\./\\\\./g\<CR>0\"9y$u:menu Buffers.\<C-R>9 :buffer \<C-R>%\<C-V>\
<CR>\<CR>"
```

For a real buffer menu, user functions should be used (see :function), but then the <> notation isn't used, which defeats using it as an example here.

Highlighting matching parens

match-parens

This example shows the use of a few advanced tricks:

- using the CursorMoved autocommand event
- using searchpairpos() to find a matching paren
- using synID() to detect whether the cursor is in a string or comment
- using :match to highlight something
- using a pattern to match a specific position in the file.

This should be put in a Vim script file, since it uses script-local variables. It skips matches in strings or comments, unless the cursor started in string or comment. This requires syntax highlighting.

A slightly more advanced version is used in the matchparen plugin.

```
let s:paren_hl_on = 0
function s:Highlight_Matching_Paren()
  if s:paren_hl_on
    match none
    let s:paren_hl_on = 0
  endif

  let c_lnum = line('.')
  let c_col = col('.')

  let c = getline(c_lnum)[c_col - 1]
  let plist = split(&matchpairs, ':\|,')
  let i = index(plist, c)
  if i < 0
    return
  endif
  if i % 2 == 0
    let s_flags = 'nW'
    let c2 = plist[i + 1]
  else
    let s_flags = 'nbW'
    let c2 = c
    let c = plist[i - 1]
  endif
  if c == '['
    let c = '\['
    let c2 = '\]'
  endif
  let s_skip ='synIDattr(synID(line("."), col("."), 0), "name") ' ..
        \ '=~?  "string\\|comment"'
  execute 'if' s_skip '| let s_skip = 0 | endif'

  let [m_lnum, m_col] = searchpairpos(c, '', c2, s_flags, s_skip)

  if m_lnum > 0 && m_lnum >= line('w0') && m_lnum <= line('w$')
    exe 'match Search /\(\%' .. c_lnum .. 'l\%' .. c_col ..
          \ 'c\)\|\(\%' .. m_lnum .. 'l\%' .. m_col .. 'c\)/'
    let s:paren_hl_on = 1
  endif
endfunction

autocmd CursorMoved,CursorMovedI * call s:Highlight_Matching_Paren()
autocmd InsertEnter * match none
```

Opening help in the current window

help-curwin

By default, help is displayed in a split window. If you prefer it opens in
the current window, try this custom :HelpCurwin command:

```
command -bar -nargs=? -complete=help HelpCurwin execute s:HelpCurwin(<q-args>)
let s:did_open_help = v:false

function s:HelpCurwin(subject) abort
  let mods = 'silent noautocmd keepalt'
  if !s:did_open_help
    execute mods .. ' help'
    execute mods .. ' helpclose'
    let s:did_open_help = v:true
  endif
  if !empty(getcompletion(a:subject, 'help'))
    execute mods .. ' edit ' .. &helpfile
    set buftype=help
  endif
  return 'help ' .. a:subject
endfunction
```

Treesitter integration

Nvim integrates the tree-sitter library for incremental parsing of buffers:
https://tree-sitter.github.io/tree-sitter/

WARNING: Treesitter support is still experimental and subject to frequent changes. This documentation may also not fully reflect the latest changes.

PARSER FILES

treesitter-parsers

Parsers are the heart of treesitter. They are libraries that treesitter will search for in the parser runtime directory.

Nvim includes these parsers:

- C
- Diff
- Lua
- Markdown
- Vimscript
- Vimdoc
- Treesitter query files ft-query-plugin

You can install more parsers manually, or with a plugin like
https://github.com/nvim-treesitter/nvim-treesitter .

Parsers are searched for as parser/{lang}.* in any 'runtimepath' directory. If multiple parsers for the same language are found, the first one is used. (NOTE: This typically implies the priority "user config > plugins > bundled".)

To load a parser from its filepath:

```
vim.treesitter.language.add('python', { path = "/path/to/python.so" })
```

Parser names are assumed to be lower case if the file system is case-sensitive.

To associate certain filetypes with a treesitter language (name of parser), use vim.treesitter.language.register(). For example, to use the xml treesitter parser for buffers with filetype svg or xslt, use:

```
vim.treesitter.language.register('xml', { 'svg', 'xslt' })
```

treesitter-parsers-wasm

If Nvim is built with ENABLE_WASMTIME, then wasm parsers can also be loaded:

```
vim.treesitter.language.add('python', { path = "/path/to/python.wasm" })
```

TREESITTER QUERIES

treesitter-query

Treesitter queries are a way to extract information about a parsed TSTree, e.g., for the purpose of highlighting. Briefly, a query consists of one or more patterns. A pattern is defined over node types in the syntax tree. A match corresponds to specific elements of the syntax tree which match a pattern. Patterns may optionally define captures and predicates. A capture allows you to associate names with a specific node in a pattern. A predicate adds arbitrary metadata and conditional data to a match.

Queries are written in a lisp-like language documented in
https://tree-sitter.github.io/tree-sitter/using-parsers/queries/1-syntax.html
Note: The predicates listed there differ from those Nvim supports. See treesitter-predicates for a complete list of predicates supported by Nvim.

Nvim looks for queries as *.scm files in a queries directory under runtimepath, where each file contains queries for a specific language and purpose, e.g., queries/lua/highlights.scm for highlighting Lua files. By default, the first query on runtimepath is used (which usually implies that user config takes precedence over plugins, which take precedence over queries bundled with Nvim). If a query should extend other queries instead of replacing them, use treesitter-query-modeline-extends.

The Lua interface is described at lua-treesitter-query.

TREESITTER QUERY PREDICATES **treesitter-predicates**

Predicates are special scheme nodes that are evaluated to conditionally capture nodes. For example, the eq? predicate can be used as follows:

```
((identifier) @variable.builtin
  (#eq? @variable.builtin "self"))
```

to only match identifier corresponding to the "self" text. Such queries can be used to highlight built-in functions or variables differently, for instance.

The following predicates are built in:

eq? treesitter-predicate-eq?

Match a string against the text corresponding to a node:

```
((identifier) @variable.builtin (#eq? @variable.builtin "self"))
((node1) @left (node2) @right (#eq? @left @right))
```

any-eq? treesitter-predicate-any-eq?

Like eq?, but for quantified patterns only one captured node must match.

match? treesitter-predicate-match?

vim-match? treesitter-predicate-vim-match?

Match a regexp against the text corresponding to a node:

```
((identifier) @constant (#match? @constant "^[A-Z_]+$"))
```

Note: The ^ and $ anchors will match the start and end of the node's text.

any-match? treesitter-predicate-any-match?

any-vim-match? treesitter-predicate-any-vim-match?

Like match?, but for quantified patterns only one captured node must match.

lua-match? treesitter-predicate-lua-match?

Match lua-patterns against the text corresponding to a node, similar to match?

any-lua-match? treesitter-predicate-any-lua-match?

Like lua-match?, but for quantified patterns only one captured node must match.

contains? treesitter-predicate-contains?

Match a string against parts of the text corresponding to a node:

```
((identifier) @foo (#contains? @foo "foo"))
((identifier) @foo-bar (#contains? @foo-bar "foo" "bar"))
```

any-contains? treesitter-predicate-any-contains?

Like contains?, but for quantified patterns only one captured node must match.

any-of? treesitter-predicate-any-of?

Match any of the given strings against the text corresponding to a node:

```
((identifier) @foo (#any-of? @foo "foo" "bar"))
```

This is the recommended way to check if the node matches one of many keywords, as it has been optimized for this.

has-ancestor? treesitter-predicate-has-ancestor?

Match any of the given node types against all ancestors of a node:

```
((identifier) @variable.builtin
  (#any-of? @variable.builtin "begin" "end")
  (#has-ancestor? @variable.builtin range_expression))
```

has-parent? treesitter-predicate-has-parent?

Match any of the given node types against the direct ancestor of a node:

```
(((field_expression
     (field_identifier) @method)) @_parent
 (#has-parent? @_parent template_method function_declarator))
```

treesitter-predicate-not

Each predicate has a not- prefixed predicate that is just the negation of
the predicate.

treesitter-predicate-all

treesitter-predicate-any

Queries can use quantifiers to capture multiple nodes. When a capture contains
multiple nodes, predicates match only if ALL nodes contained by the capture
match the predicate. Some predicates (eq?, match?, lua-match?,
contains?) accept an any- prefix to instead match if ANY of the nodes
contained by the capture match the predicate.

As an example, consider the following Lua code:

```
-- TODO: This is a
-- very long
-- comment (just imagine it)
```

using the following predicated query:

```
(((comment)+ @comment)
 (#match? @comment "TODO"))
```

This query will not match because not all of the nodes captured by @comment
match the predicate. Instead, use:

```
(((comment)+ @comment)
 (#any-match? @comment "TODO"))
```

Further predicates can be added via vim.treesitter.query.add_predicate().
Use vim.treesitter.query.list_predicates() to list all available predicates.

TREESITTER QUERY DIRECTIVES

treesitter-directives

Treesitter directives store metadata for a node or match and perform side
effects. For example, the set! directive sets metadata on the match or node:

```
((identifier) @foo (#set! type "parameter"))
```

The following directives are built in:

set!

treesitter-directive-set!

Sets key/value metadata for a specific match or capture. Value is
accessible as either metadata[key] (match specific) or
metadata[capture_id][key] (capture specific).

Parameters:
{capture_id} (optional)
{key}
{value}

Examples:

```
((identifier) @foo (#set! @foo kind "parameter"))
((node1) @left (node2) @right (#set! type "pair"))
((codeblock) @markup.raw.block (#set! priority 90))
```

offset!

treesitter-directive-offset!

Takes the range of the captured node and applies an offset. This will
set a new range in the form of a list like { {start_row}, {start_col},
{end_row}, {end_col} } for the captured node with capture_id as
metadata[capture_id].offset. This offset will be applied to the
range returned in vim.treesitter.get_range(). Useful for
treesitter-language-injections.

Parameters:
{capture_id}
{start_row}
{start_col}
{end_row}
{end_col}

Example:

```
((identifier) @constant (#offset! @constant 0 1 0 -1))
```

gsub!

treesitter-directive-gsub!

Transforms the content of the node using a lua-pattern. This will set
a new metadata[capture_id].text.

Parameters:
{capture_id}
{pattern}
{replacement}

Example:

```
(#gsub! @_node ".*%.(.*)" "%1")
```

trim! treesitter-directive-trim!

Trims whitespace from the node. Sets a new metadata[capture_id].range. Takes a capture ID and, optionally, four integers to customize trimming behavior (1 meaning trim, 0 meaning don't trim). When only given a capture ID, trims blank lines (lines that contain only whitespace, or are empty) from the end of the node (for backwards compatibility). Can trim all whitespace from both sides of the node if parameters are given.

Examples:

```
; only trim blank lines from the end of the node
; (equivalent to (#trim! @fold 0 0 1 0))
(#trim! @fold)

; trim blank lines from both sides of the node
(#trim! @fold 1 0 1 0)

; trim all whitespace around the node
(#trim! @fold 1 1 1 1)
```

Parameters:
{capture_id}
{trim_start_linewise}
{trim_start_charwise}
{trim_end_linewise} (default 1 if only given {capture_id})
{trim_end_charwise}

Further directives can be added via vim.treesitter.query.add_directive().
Use vim.treesitter.query.list_directives() to list all available directives.

TREESITTER QUERY MODELINES treesitter-query-modeline

Nvim supports to customize the behavior of the queries using a set of "modelines", that is comments in the queries starting with ;. Here are the currently supported modeline alternatives:

inherits: {lang}... treesitter-query-modeline-inherits

Specifies that this query should inherit the queries from {lang}. This will recursively descend in the queries of {lang} unless wrapped in parentheses: ({lang}).
Note: This is meant to be used to include queries from another language. If you want your query to extend the queries of the same language, use extends.

extends treesitter-query-modeline-extends

Specifies that this query should be used as an extension for the query, i.e. that it should be merged with the others.
Note: The order of the extensions, and the query that will be used as a base depends on your 'runtimepath' value.

Note: These modeline comments must be at the top of the query, but can be repeated, for example, the following two modeline blocks are both valid:

```
;; inherits: typescript,jsx
;; extends
```

```
;; extends
;;
;; inherits: css
```

TREESITTER SYNTAX HIGHLIGHTING treesitter-highlight

Syntax highlighting is specified through queries named highlights.scm, which match a TSNode in the parsed TSTree to a capture that can be assigned a highlight group. For example, the query

```
(parameters (identifier) @variable.parameter)
```

matches any identifier node inside a function parameters node to the capture named @variable.parameter. For example, for a Lua code

```
function f(foo, bar) end
```

which will be parsed as (see :InspectTree):

```
(function_declaration ; [1:1 - 24]
  name: (identifier) ; [1:10 - 10]
  parameters: (parameters ; [1:11 - 20]
    name: (identifier) ; [1:12 - 14]
    name: (identifier))) ; [1:17 - 19]
```

the above query will highlight foo and bar as @variable.parameter.

It is also possible to match literal expressions (provided the parser returns them):

```
[
  "if"
  "else"
] @keyword.conditional
```

Assuming a suitable parser and highlights.scm query is found in runtimepath, treesitter highlighting for the current buffer can be enabled simply via vim.treesitter.start().

treesitter-highlight-groups

The capture names, prefixed with @, are directly usable as highlight groups. For many commonly used captures, the corresponding highlight groups are linked to Nvim's standard highlight-groups by default (e.g., @comment links to Comment) but can be overridden in colorschemes.

A fallback system is implemented, so that more specific groups fallback to more generic ones. For instance, in a language that has separate doc comments (e.g., c, java, etc.), @comment.documentation could be used. If this group is not defined, the highlighting for an ordinary @comment is used. This way, existing color schemes already work out of the box, but it is possible to add more specific variants for queries that make them available.

As an additional rule, capture highlights can always be specialized by language, by appending the language name after an additional dot. For instance, to highlight comments differently per language:

```
hi @comment.c guifg=Blue
hi @comment.lua guifg=DarkBlue
hi link @comment.documentation.java String
```

The following is a list of standard captures used in queries for Nvim, highlighted according to the current colorscheme (use :Inspect on one to see the exact definition):

```
@variable                       various variable names
@variable.builtin               built-in variable names (e.g. this, self)
@variable.parameter             parameters of a function
@variable.parameter.builtin     special parameters (e.g. _, it)
@variable.member                object and struct fields

@constant               constant identifiers
@constant.builtin       built-in constant values
@constant.macro         constants defined by the preprocessor

@module                 modules or namespaces
@module.builtin         built-in modules or namespaces
@label                  GOTO and other labels (e.g. label: in C), including heredoc labels

@string                 string literals
@string.documentation   string documenting code (e.g. Python docstrings)
@string.regexp          regular expressions
@string.escape          escape sequences
@string.special         other special strings (e.g. dates)
@string.special.symbol  symbols or atoms
@string.special.path    filenames
@string.special.url     URIs (e.g. hyperlinks)

@character              character literals
@character.special      special characters (e.g. wildcards)

@boolean                boolean literals
@number                 numeric literals
@number.float           floating-point number literals

@type                   type or class definitions and annotations
@type.builtin           built-in types
@type.definition        identifiers in type definitions (e.g. typedef <type> <identifier> in C)

@attribute              attribute annotations (e.g. Python decorators, Rust lifetimes)
@attribute.builtin      builtin annotations (e.g. @property in Python)
@property               the key in key/value pairs

@function               function definitions
@function.builtin       built-in functions
@function.call          function calls
@function.macro         preprocessor macros

@function.method        method definitions
@function.method.call   method calls

@constructor            constructor calls and definitions
@operator               symbolic operators (e.g. +, *)

@keyword                keywords not fitting into specific categories
@keyword.coroutine      keywords related to coroutines (e.g. go in Go, async/await in Python)
@keyword.function       keywords that define a function (e.g. func in Go, def in Python)
@keyword.operator       operators that are English words (e.g. and, or)
@keyword.import         keywords for including or exporting modules (e.g. import, from in Python)
@keyword.type           keywords describing namespaces and composite types (e.g. struct, enum)
@keyword.modifier       keywords modifying other constructs (e.g. const, static, public)
@keyword.repeat         keywords related to loops (e.g. for, while)
@keyword.return         keywords like return and yield
@keyword.debug          keywords related to debugging
@keyword.exception      keywords related to exceptions (e.g. throw, catch)

@keyword.conditional         keywords related to conditionals (e.g. if, else)
@keyword.conditional.ternary ternary operator (e.g. ?, :)
```

```
@keyword.directive           various preprocessor directives and shebangs
@keyword.directive.define    preprocessor definition directives

@punctuation.delimiter  delimiters (e.g. ;, ., ,)
@punctuation.bracket    brackets (e.g. (), {}, [])
@punctuation.special    special symbols (e.g. {} in string interpolation)

@comment                line and block comments
@comment.documentation  comments documenting code

@comment.error          error-type comments (e.g. ERROR, FIXME, DEPRECATED)
@comment.warning        warning-type comments (e.g. WARNING, FIX, HACK)
@comment.todo           todo-type comments (e.g. TODO, WIP)
@comment.note           note-type comments (e.g. NOTE, INFO, XXX)

@markup.strong          bold text
@markup.italic          italic text
@markup.strikethrough   struck-through text
@markup.underline       underlined text (only for literal underline markup!)

@markup.heading         headings, titles (including markers)
@markup.heading.1       top-level heading
@markup.heading.2       section heading
@markup.heading.3       subsection heading
@markup.heading.4       and so on
@markup.heading.5       and so forth
@markup.heading.6       six levels ought to be enough for anybody

@markup.quote           block quotes
@markup.math            math environments (e.g. $ ... $ in LaTeX)

@markup.link            text references, footnotes, citations, etc.
@markup.link.label      link, reference descriptions
@markup.link.url        URL-style links

@markup.raw             literal or verbatim text (e.g. inline code)
@markup.raw.block       literal or verbatim text as a stand-alone block

@markup.list            list markers
@markup.list.checked    checked todo-style list markers
@markup.list.unchecked  unchecked todo-style list markers

@diff.plus              added text (for diff files)
@diff.minus             deleted text (for diff files)
@diff.delta             changed text (for diff files)

@tag                    XML-style tag names (e.g. in XML, HTML, etc.)
@tag.builtin            builtin tag names (e.g. HTML5 tags)
@tag.attribute          XML-style tag attributes
@tag.delimiter          XML-style tag delimiters
```

treesitter-highlight-spell

The special @spell capture can be used to indicate that a node should be spell checked by Nvim's builtin spell checker. For example, the following capture marks comments as to be checked:

```
(comment) @spell
```

There is also @nospell which disables spellchecking regions with @spell.

treesitter-highlight-conceal

Treesitter highlighting supports conceal via the conceal and conceal_lines metadata. By convention, nodes to be concealed are captured as @conceal, but any capture can be used. For example, the following query can be used to hide code block delimiters in Markdown:

```
((fenced_code_block_delimiter) @conceal (#set! conceal ""))
```

It is also possible to replace a node with a single character, which (unlike legacy syntax) can be given a custom highlight. For example, the following (ill-advised) query replaces the != operator by a Unicode glyph, which is still highlighted the same as other operators:

```
"!=" @operator (#set! conceal "≠")
```

To conceal an entire line (do not draw it at all), a query with conceal_lines metadata can be used:

```
((comment) @comment @spell
  (#set! conceal_lines ""))
```

Conceals specified in this way respect 'conceallevel' and 'concealcursor'.

Note that although you can use any string for conceal, only the first character will be used:

```
; identifiers will be concealed with 'f'.
((identifier) @conceal (#set! conceal "foo"))
```

To remove a conceal highlight, you can use the @noconceal capture to override any existing conceal highlight set for the range.

```
((identifier) @noconceal)
```

treesitter-highlight-priority

Treesitter uses nvim_buf_set_extmark() to set highlights with a default priority of 100. This enables plugins to set a highlighting priority lower or higher than treesitter. It is also possible to change the priority of an individual query pattern manually by setting its "priority" metadata attribute:

```
((super_important_node) @superimportant (#set! priority 105))
```

treesitter-highlight-commentstring

Treesitter highlighting supports finer-grained 'commentstring's, used by the built-in commenting plugin. When the cursor is within a node that sets the bo.commentstring metadata property (treesitter-directive-set!), that property defines the comment delimiter (where "innermost wins"). This is useful for languages like JSX that have different comment syntax depending on the code region, for example:

```
((jsx_element) @_tag (#set! @_tag bo.commentstring "{/* %s */}"))
```

When multiple captures set this metadata over a region, only the innermost (most specific) one is applied to a given area.

TREESITTER LANGUAGE INJECTIONS

treesitter-language-injections

Note the following information is adapted from:
https://tree-sitter.github.io/tree-sitter/3-syntax-highlighting.html#language-injection

Some source files contain code written in multiple different languages. Examples include:

- HTML files, which can contain JavaScript inside of <script> tags and CSS inside of <style> tags
- ERB files, which contain Ruby inside of <% %> tags, and HTML outside of those tags
- PHP files, which can contain HTML between the <php tags
- JavaScript files, which contain regular expression syntax within regex literals
- Ruby, which can contain snippets of code inside of heredoc literals, where the heredoc delimiter often indicates the language
- Lua, which can contain snippets of Vimscript inside vim.cmd() calls.
- Vimscript, which can contain snippets of Lua inside :lua-heredoc blocks.

All of these examples can be modeled in terms of a parent syntax tree and one or more injected syntax trees, which reside inside of certain nodes in the parent tree. The language injection query allows you to specify these “injections” using the following captures:

- @injection.content - indicates that the captured node should have its contents re-parsed using another language. If there are multiple @injection.content captures in one pattern, all ranges will be collected and parsed as one tree. This allows query authors to create "scoped" injections with injection query quantifiers.
- @injection.language - indicates that the captured node’s text may contain the name of a language that should be used to re-parse the @injection.content.
- @injection.filename - indicates that the captured node’s text may contain a filename; the corresponding filetype is then looked-up up via vim.filetype.match() and treated as the name of a language that should be used to re-parse the @injection.content.

The language injection behavior can also be configured by some properties associated with patterns:

- injection.language - can be used to hard-code the name of a specific

language.
- injection.combined - indicates that all of the matching nodes in the tree should have their content parsed as one nested document.
- injection.include-children - indicates that the @injection.content node's entire text should be re-parsed, including the text of its child nodes. By default, child nodes' text will be excluded from the injected document.
- injection.self - indicates that the node's text should be parsed with the same language as the node's LanguageTree.
- injection.parent - indicates that the captured node's text should be parsed with the same language as the node's parent LanguageTree.

Injection queries are currently run over the entire buffer, which can be slow for large buffers. To disable injections for, e.g., c, just place an empty queries/c/injections.scm file in your 'runtimepath'.

DEFAULTS

treesitter-defaults

treesitter-incremental-selection

v_an

an — Selects [count]th parent node.
If buffer has no treesitter parser, falls back to vim.lsp.buf.selection_range().

v_in

in — Selects [count]th previous (or first) child node.
If buffer has no treesitter parser, falls back to vim.lsp.buf.selection_range().

v_]n

]n — Selects [count]th next node.

v_[n

[n — Selects [count]th previous node.

v_]N

]N — Expands selection to [count]th next node.

v_[N

[N — Expands selection to [count]th previous node.

VIM.TREESITTER

lua-treesitter **vim.treesitter**

The remainder of this document is a reference manual for the vim.treesitter Lua module, which is the main interface for Nvim's treesitter integration. Most of the following content is automatically generated from the function documentation.

vim.treesitter.language_version

The latest parser ABI version that is supported by the bundled treesitter library.

vim.treesitter.minimum_language_version

The earliest parser ABI version that is supported by the bundled treesitter library.

TREESITTER TREES

treesitter-tree **TSTree**

A "treesitter tree" represents the parsed contents of a buffer, which can be used to perform further analysis. It is a userdata reference to an object held by the treesitter library.

An instance TSTree of a treesitter tree supports the following methods.

TSTree:copy()

TSTree:copy()

Returns a copy of the TSTree.

Return:
(TSTree)

TSTree:root()

TSTree:root()

Return the root node of this tree.

Return:
(TSNode)

TREESITTER NODES

treesitter-node **TSNode**

A "treesitter node" represents one specific element of the parsed contents of a buffer, which can be captured by a treesitter-query for, e.g., highlighting. It is a userdata reference to an object held by the treesitter library.

An instance TSNode of a treesitter node supports the following methods.

TSNode:byte_length() TSNode:byte_length()

Return the number of bytes spanned by this node.

Return:
(integer)

TSNode:child({index}) TSNode:child()

Get the node's child at the given {index}, where zero represents the first child.

Parameters:
- {index} (integer)

Return:
(TSNode?)

TSNode:child_count() TSNode:child_count()

Get the node's number of children.

Return:
(integer)

TSNode:child_with_descendant()

TSNode:child_with_descendant({descendant})
Get the node's child that contains {descendant} (includes {descendant}).

For example, with the following node hierarchy:

```
a -> b -> c

a:child_with_descendant(c) == b
a:child_with_descendant(b) == b
a:child_with_descendant(a) == nil
```

Parameters:
- {descendant} (TSNode)

Return:
(TSNode?)

TSNode:descendant_for_range()

TSNode:descendant_for_range({start_row}, {start_col}, {end_row}, {end_col})
Get the smallest node within this node that spans the given range of (row, column) positions

Parameters:
- {start_row} (integer)
- {start_col} (integer)
- {end_row} (integer)
- {end_col} (integer)

Return:
(TSNode?)

TSNode:end_() TSNode:end_()

Get the node's end position. Return three values: the row, column and total byte count (all zero-based).

Return (multiple):
(integer)
(integer)
(integer)

TSNode:equal({node}) TSNode:equal()

Check if {node} refers to the same node within the same tree.

Parameters:
- {node} (TSNode)

Return:

(boolean)

TSNode:extra() TSNode:extra()

Check if the node is extra. Extra nodes represent things like comments, which are not required by the grammar but can appear anywhere.

Return:
(boolean)

TSNode:field({name}) TSNode:field()

Returns a list of all the node's children that have the given field name.

Parameters:
- {name} (string)

Return:
(TSNode[])

TSNode:has_changes() TSNode:has_changes()

Check if a syntax node has been edited.

Return:
(boolean)

TSNode:has_error() TSNode:has_error()

Check if the node is a syntax error or contains any syntax errors.

Return:
(boolean)

TSNode:id() TSNode:id()

Get a unique identifier for the node inside its own tree.

No guarantees are made about this identifier's internal representation, except for being a primitive Lua type with value equality (so not a table). Presently it is a (non-printable) string.

Note: The id is not guaranteed to be unique for nodes from different trees.

Return:
(string)

TSNode:iter_children() TSNode:iter_children()

Iterates over all the direct children of {TSNode}, regardless of whether they are named or not. Returns the child node plus the eventual field name corresponding to this child node.

Return:
(fun(): TSNode, string)

TSNode:missing() TSNode:missing()

Check if the node is missing. Missing nodes are inserted by the parser in order to recover from certain kinds of syntax errors.

Return:
(boolean)

TSNode:named() TSNode:named()

Check if the node is named. Named nodes correspond to named rules in the grammar, whereas anonymous nodes correspond to string literals in the grammar.

Return:
(boolean)

TSNode:named_child({index}) TSNode:named_child()

Get the node's named child at the given {index}, where zero represents the first named child.

Parameters:
- {index} (integer)

Return:
(TSNode?)

TSNode:named_child_count() TSNode:named_child_count()

Get the node's number of named children.

Return:
(integer)

TSNode:named_children() TSNode:named_children()

Returns a list of the node's named children.

Return:
(TSNode[])

TSNode:named_descendant_for_range()

TSNode:named_descendant_for_range({start_row}, {start_col}, {end_row},
                                  {end_col})
    Get the smallest named node within this node that spans the given range of
    (row, column) positions

Parameters:
- {start_row} (integer)
- {start_col} (integer)
- {end_row}   (integer)
- {end_col}   (integer)

Return:
        (TSNode?)

TSNode:next_named_sibling() TSNode:next_named_sibling()

    Get the node's next named sibling.

Return:
        (TSNode?)

TSNode:next_sibling() TSNode:next_sibling()

    Get the node's next sibling.

Return:
        (TSNode?)

TSNode:parent() TSNode:parent()

    Get the node's immediate parent. Prefer TSNode:child_with_descendant()
    for iterating over the node's ancestors.

Return:
        (TSNode?)

TSNode:prev_named_sibling() TSNode:prev_named_sibling()

    Get the node's previous named sibling.

Return:
        (TSNode?)

TSNode:prev_sibling() TSNode:prev_sibling()

    Get the node's previous sibling.

Return:
        (TSNode?)

TSNode:range({include_bytes}) TSNode:range()

    Get the range of the node.

    Return four or six values:
- start row
- start column
- start byte (if {include_bytes} is true)
- end row
- end column
- end byte (if {include_bytes} is true)

Parameters:
- {include_bytes} (false?)

Overloads:
- fun(self: TSNode, include_bytes: true): integer, integer, integer, integer, integer, integer

Return (multiple):
        (integer)
        (integer)
        (integer)
        (integer)

TSNode:sexpr() TSNode:sexpr()

Get an S-expression representing the node as a string.

Return:
(string)

TSNode:start() TSNode:start()

Get the node's start position. Return three values: the row, column and total byte count (all zero-based).

Return (multiple):
(integer)
(integer)
(integer)

TSNode:symbol() TSNode:symbol()

Get the node's type as a numerical id.

Return:
(integer)

TSNode:tree() TSNode:tree()

Get the TSTree of the node.

Return:
(TSTree)

TSNode:type() TSNode:type()

Get the node's type as a string.

Return:
(string)

Lua module: vim.treesitter lua-treesitter-core

foldexpr({lnum}) vim.treesitter.foldexpr()

Returns the fold level for {lnum} in the current buffer. Can be set directly to 'foldexpr':

```
vim.wo.foldexpr = vim.treesitter.foldexpr
```

Attributes:
Since: 0.9.0

Parameters:
- {lnum} (integer?) Line number to calculate fold level for

Return:
(string)

get_captures({buf}, {start}, {stop}) vim.treesitter.get_captures()

Returns highlight captures at a position, on a line, or overlapping a range.

{start} and {stop} can be a row number or a {row, col} pair. All rows and columns are 0-based; columns are byte offsets.

Without {stop}, a row number queries the whole line and a {row, col} pair queries one position.

With {stop}, return captures that overlap the range. The stop position is excluded; a row number excludes that row. Empty ranges return no captures.

Each result includes the full captured node range, which may extend outside the requested range. Returns an empty list if treesitter highlighting is not active in {buf}.

Attributes:
Since: 0.13.0

Parameters:
- {buf} (integer) Buffer number (0 for current buffer)
- {start} (integer|[integer,integer]) Start row or {row, col}
- {stop} (integer|[integer,integer]?) Stop row or {row, col}

(excluded)

Return:
(table[]) A list of objects with the following fields:
- {capture} (string) Capture name
- {col} (integer) Start column
- {end_col} (integer) End column (excluded)

- {end_row} (integer) End row
- {id} (integer) 1-based capture id in the query
- {lang} (string) Language name
- {metadata} (vim.treesitter.query.TSMetadata) Query metadata, e.g.

  priority or conceal
- {pattern_id} (integer) 1-based id of the matched pattern in the

  query
- {row} (integer) Start row

get_captures_at_cursor({win}) vim.treesitter.get_captures_at_cursor()

Returns a list of highlight capture names under the cursor

Parameters:
- {win} (integer?) window-ID or 0 for current window (default)

Return:

(string[]) List of capture names

get_node({opts}) vim.treesitter.get_node()

Returns the smallest named node at the given position

NOTE: Calling this on an unparsed tree can yield an invalid node. If the tree is not known to be parsed by, e.g., an active highlighter, parse the tree first via

```
vim.treesitter.get_parser(bufnr):parse(range)
```

Parameters:
- {opts} (table?) Optional keyword arguments:
  - {bufnr} (integer?) Buffer number (nil or 0 for current

    buffer)
  - {ignore_injections} (boolean?, default: true) Ignore

    injected languages
  - {include_anonymous} (boolean?, default: false) Include

    anonymous nodes
  - {lang} (string?, default: from 'filetype') Parser

    language.
  - {pos} ([integer, integer]?, default: window-local cursor)

    0-indexed (row, col) tuple. Required if {bufnr} is not the current buffer

Return:

(TSNode?) Node at the given position

get_node_range({node_or_range}) vim.treesitter.get_node_range()

Returns the node's range or an unpacked range table

Parameters:
- {node_or_range} (TSNode|Range4) Node or table of positions

Return (multiple):

(integer) start_row
(integer) start_col (byte offset)
(integer) end_row
(integer) end_col (byte offset)

vim.treesitter.get_node_text()

get_node_text({node}, {source}, {opts})

Gets the text corresponding to a given node

Parameters:
- {node} (TSNode)

- {source} (integer|string) Buffer or string from which the {node} is extracted
- {opts} (table?) Optional parameters.
  - metadata (table) Metadata of a specific capture. This would be set to metadata[capture_id] when using vim.treesitter.query.add_directive().

Return:
(string)

get_parser({buf}, {lang}, {opts}) vim.treesitter.get_parser()

Returns the parser for a specific buffer and attaches it to the buffer

If needed, this will create the parser.

If no parser can be created, nil (and an error message) is returned.

Parameters:
- {buf} (integer?) (default: current buffer) Buffer the parser should be tied to
- {lang} (string?) (default: from 'filetype') Language of this parser
- {opts} (table?) Options to pass to the created language tree

Return (multiple):
(vim.treesitter.LanguageTree?) object to use for parsing
(string?) error message, if applicable

get_range({node}, {source}, {metadata}) vim.treesitter.get_range()

Get the range of a TSNode. Can also supply {source} and {metadata} to get the range with directives applied.

Parameters:
- {node} (TSNode)
- {source} (integer|string?) Buffer or string from which the {node} is extracted
- {metadata} (vim.treesitter.query.TSMetadata?)

Return:
(table) A table with the following fields:
- {[1]} (integer) start row
- {[2]} (integer) start column
- {[3]} (integer) start bytes
- {[4]} (integer) end row
- {[5]} (integer) end column
- {[6]} (integer) end bytes

vim.treesitter.get_string_parser()

get_string_parser({str}, {lang}, {opts})
Returns a string parser

Parameters:
- {str} (string) Text to parse
- {lang} (string) Language of this string
- {opts} (table?) Options to pass to the created language tree

Return:
(vim.treesitter.LanguageTree) object to use for parsing

inspect_tree({opts}) vim.treesitter.inspect_tree()

Open a window that displays a textual representation of the nodes in the language tree.

While in the window, press "a" to toggle display of anonymous nodes, "I" to toggle the display of the source language of each node, "o" to toggle the query editor, and press <Enter> to jump to the node under the cursor in the source buffer. Folding also works (try zo, zc, etc.).

Can also be shown with :InspectTree. :InspectTree

Attributes:
Since: 0.9.0

Parameters:
- {opts} (table?) Optional options table with the following possible keys:
  - bufnr (integer|nil): Buffer to draw the tree into. If omitted, a new buffer is created.
  - command (string|nil): Vimscript command to create the window. Default value is "60vnew". Only used when {winid} is nil.
  - lang (string|nil): The language of the source buffer. If omitted, detect from the filetype of the source buffer.
  - title (string|fun(bufnr:integer):string|nil): Title of the window. If a function, it accepts the buffer number of the source buffer as its only argument and should return a string.
  - winid (integer|nil): Window id to display the tree buffer in. If omitted, a new window is created with {command}.

is_ancestor({dest}, {source}) vim.treesitter.is_ancestor()

Determines whether a node is the ancestor of another

Parameters:
- {dest} (TSNode) Possible ancestor
- {source} (TSNode) Possible descendant

Return:
(boolean) True if {dest} is an ancestor of {source}

vim.treesitter.is_in_node_range()

is_in_node_range({node}, {line}, {col})
Determines whether (line, col) position is in node range

Parameters:
- {node} (TSNode) defining the range
- {line} (integer) Line (0-based)
- {col} (integer) Column (0-based)

Return:
(boolean) True if the position is in node range

node_contains({node}, {range}) vim.treesitter.node_contains()

Determines if a node contains a range

Parameters:
- {node} (TSNode)
- {range} (table)

Return:
(boolean) True if the {node} contains the {range}

select({target}, {count}) vim.treesitter.select()

Starts or adjusts a Visual selection at cursor, based on tree nodes. The target parameter decides the selection behavior.

Parameters:
- {target} ('parent'|'child'|'next'|'prev'|'extend_next'|'extend_prev')

  Decides the selection behavior.
- {count} (integer?) (default: 1) Expand or adjust the selection

  this many times.

start({buf}, {lang}) vim.treesitter.start()

Starts treesitter highlighting for a buffer

Can be used in an ftplugin or FileType autocommand.

Note: By default, disables regex syntax highlighting, which may be required for some plugins. In this case, add vim.bo.syntax = 'ON' after the call to start.

Note: By default, the highlighter parses code asynchronously, using a segment time of 3ms.

Example:

```
vim.api.nvim_create_autocmd( 'FileType', { pattern = 'tex',
    callback = function(ev)
        vim.treesitter.start(ev.buf, 'latex')
        vim.bo[ev.buf].syntax = 'ON'  -- only if additional legacy syntax is needed
    end
})
```

Parameters:
- {buf} (integer?) (default: current buffer) Buffer to be

  highlighted
- {lang} (string?) (default: from 'filetype') Language of the parser

stop({buf}) vim.treesitter.stop()

Stops treesitter highlighting for a buffer

Parameters:
- {buf} (integer?) (default: current buffer) Buffer to stop

  highlighting

Lua module: vim.treesitter.language treesitter-language

add({lang}, {opts}) vim.treesitter.language.add()

Load parser with name {lang}

Parsers are searched in the parser runtime directory, or the provided {path}. Can be used to check for available parsers before enabling treesitter features, e.g.,

```
if vim.treesitter.language.add('markdown') then
  vim.treesitter.start(bufnr, 'markdown')
end
```

Parameters:
- {lang} (string) Name of the parser (alphanumerical and _ only)
- {opts} (table?) Options:
  - {path}? (string) Optional path the parser is located at
  - {symbol_name}? (string) Internal symbol name for the

    language to load

Return (multiple):
(boolean?) True if parser is loaded
(string?) Error if parser cannot be loaded

get_filetypes({lang}) vim.treesitter.language.get_filetypes()

Returns the filetypes for which a parser named {lang} is used.

The list includes {lang} itself plus all filetypes registered via vim.treesitter.language.register().

Parameters:
- {lang} (string) Name of parser

Return:

(string[]) filetypes

get_lang({filetype}) vim.treesitter.language.get_lang()

Returns the language name to be used when loading a parser for {filetype}.

If no language has been explicitly registered via vim.treesitter.language.register(), default to {filetype}. For composite filetypes like html.glimmer, only the main filetype is returned.

Parameters:
- {filetype} (string)

Return:
(string?)

inspect({lang}) vim.treesitter.language.inspect()

Inspects the provided language.

Inspecting provides some useful information on the language like ABI version, parser state count (a measure of parser complexity), node and field names, and whether the language came from a WASM module.

Node names are returned in a table mapping each node name to a boolean indicating whether or not the node is named (i.e., not anonymous). Anonymous nodes are surrounded with double quotes (").

For ABI 15 parsers, also show parser metadata (major, minor, patch version) and a table of supertypes with their respective subtypes.

Parameters:
- {lang} (string) Language

Return:
(TSLangInfo)

register({lang}, {filetype}) vim.treesitter.language.register()

Register a parser named {lang} to be used for {filetype}(s).

Note: this adds or overrides the mapping for {filetype}, any existing mappings from other filetypes to {lang} will be preserved.

Parameters:
- {lang} (string) Name of parser
- {filetype} (string|string[]) Filetype(s) to associate with lang

Lua module: vim.treesitter.languagetree treesitter-languagetree

A LanguageTree contains a tree of parsers: the root treesitter parser for {lang} and any "injected" language parsers, which themselves may inject other languages, recursively. For example a Lua buffer containing some Vimscript commands needs multiple parsers to fully understand its contents.

To create a LanguageTree (parser object) for a given buffer and language, use:

```
local parser = vim.treesitter.get_parser(bufnr, lang)
```

(where bufnr=0 means current buffer). lang defaults to 'filetype'. **Note:** currently the parser is retained for the lifetime of a buffer but this may change; a plugin should keep a reference to the parser object if it wants incremental updates.

Whenever you need to access the current syntax tree, parse the buffer:

```
local tree = parser:parse({ start_row, end_row })
```

This returns a table of immutable treesitter-tree objects representing the current state of the buffer. When the plugin wants to access the state after a (possible) edit it must call parse() again. If the buffer wasn't edited, the same tree will be returned again without extra work. If the buffer was parsed before, incremental parsing will be done of the changed parts.

Note: To use the parser directly inside a nvim_buf_attach() Lua callback, you must call vim.treesitter.get_parser() before you register your callback. But preferably parsing shouldn't be done directly in the change callback anyway as they will be very frequent. Rather a plugin that does any kind of analysis on a tree should use a timer to throttle too frequent updates.

LanguageTree:children() LanguageTree:children()

Returns a map of language to child tree.

Return:
(table<string,vim.treesitter.LanguageTree>)

LanguageTree:contains({range}) LanguageTree:contains()

Determines whether {range} is contained in the LanguageTree.

Parameters:

- {range} (table) A table with the following fields:
    - {[1]} (integer) start row
    - {[2]} (integer) start column
    - {[3]} (integer) end row
    - {[4]} (integer) end column

Return:
    (boolean)

LanguageTree:destroy() LanguageTree:destroy()

Destroys this LanguageTree and all its children.

Any cleanup logic should be performed here.

Note: This DOES NOT remove this tree from a parent. Instead, remove_child must be called on the parent to remove it.

LanguageTree:for_each_tree({fn}) LanguageTree:for_each_tree()

Invokes the callback for each LanguageTree recursively.

Note: This includes the invoking tree's child trees as well.

Parameters:
- {fn} (fun(tree: TSTree, ltree: vim.treesitter.LanguageTree))

LanguageTree:included_regions() LanguageTree:included_regions()

Gets the set of included regions managed by this LanguageTree. This can be different from the regions set by injection query, because a partial LanguageTree:parse() drops the regions outside the requested range. Each list represents a range in the form of { {start_row}, {start_col}, {start_bytes}, {end_row}, {end_col}, {end_bytes} }.

Return:
    (table<integer, Range6[]>)

LanguageTree:invalidate({reload}) LanguageTree:invalidate()

Invalidates this parser and its children.

Should only be called when the tracked state of the LanguageTree is not valid against the parse tree in treesitter. Doesn't clear filesystem cache. Called often, so needs to be fast.

Parameters:
- {reload} (boolean?)

LanguageTree:is_valid()

LanguageTree:is_valid({exclude_children}, {range})
Returns whether this LanguageTree is valid, i.e., LanguageTree:trees() reflects the latest state of the source. If invalid, user should call LanguageTree:parse().

Parameters:
- {exclude_children} (boolean?) whether to ignore the validity of children (default false)
- {range} (Range|Range[]?) range (or list of ranges, sorted by starting point in ascending order) to check for validity

Return:
    (boolean)

LanguageTree:lang() LanguageTree:lang()

Gets the language of this tree node.

Return:
    (string)

LanguageTree:language_for_range()

LanguageTree:language_for_range({range})
Gets the appropriate language that contains {range}.

Parameters:
- {range} (table) A table with the following fields:

- {[1]} (integer) start row
- {[2]} (integer) start column
- {[3]} (integer) end row
- {[4]} (integer) end column

Return:
(vim.treesitter.LanguageTree) tree Managing {range}

LanguageTree:named_node_for_range()

LanguageTree:named_node_for_range({range}, {opts})
Gets the smallest named node that contains {range}.

Parameters:
- {range} (table) A table with the following fields:
  - {[1]} (integer) start row
  - {[2]} (integer) start column
  - {[3]} (integer) end row
  - {[4]} (integer) end column
- {opts} (table?) A table with the following fields:
  - {ignore_injections}? (boolean, default: true) Ignore injected languages

Return:
(TSNode?)

LanguageTree:node_for_range()

LanguageTree:node_for_range({range}, {opts})
Gets the smallest node that contains {range}.

Parameters:
- {range} (table) A table with the following fields:
  - {[1]} (integer) start row
  - {[2]} (integer) start column
  - {[3]} (integer) end row
  - {[4]} (integer) end column
- {opts} (table?) A table with the following fields:
  - {ignore_injections}? (boolean, default: true) Ignore injected languages

Return:
(TSNode?)

LanguageTree:parent()

Returns the parent tree. nil for the root tree.

Return:
(vim.treesitter.LanguageTree?)

LanguageTree:parse({range}, {on_parse})

Recursively parse all regions in the language tree using treesitter-parsers for the corresponding languages and run injection queries on the parsed trees to determine whether child trees should be created and parsed.

Any region with empty range ({}, typically only the root tree) is always parsed; otherwise (typically injections) only if it intersects {range} (or

if {range} is true).

Parameters:

- {range} (boolean|Range|Range[]?) Parse this range (or list of ranges, sorted by starting point in ascending order) in the parser's source. Set to true to run a complete parse of the source (**Note:** Can be slow!) Set to false|nil to only parse regions with empty ranges (typically only the root tree without injections).
- {on_parse} (fun(err?: string, trees?: table<integer, TSTree>)?) Function invoked when parsing completes. When omitted, parsing is synchronous and always returns the trees. When provided and vim.g._ts_force_sync_parsing is not set, parsing will run asynchronously. The first argument to the function is a string representing the error type, in case of a failure (currently only possible for timeouts). The second argument is the list of trees returned by the parse (upon success), or nil if the parse timed out (determined by 'redrawtime').

  If parsing was still able to finish synchronously (within 3ms), parse() returns the list of trees. Otherwise, it returns nil.

Overloads:

- fun(self: vim.treesitter.LanguageTree, range?: boolean|Range|Range[]): table<integer, TSTree>

Return:

(table<integer, TSTree>?)

LanguageTree:register_cbs()

LanguageTree:register_cbs({cbs}, {recursive})

Registers callbacks for the LanguageTree.

Parameters:

- {cbs} (table<TSCallbackNameOn,function>) An nvim_buf_attach()-like table argument with the following handlers:
  - on_bytes : see nvim_buf_attach().
  - on_changedtree : a callback that will be called every time the tree has syntactical changes. It will be passed two arguments: a table of the ranges (as node ranges) that changed and the changed tree.
  - on_child_added : emitted when a child is added to the tree.
  - on_child_removed : emitted when a child is removed from the tree.
  - on_detach : emitted when the buffer is detached, see nvim_buf_detach_event. Takes one argument, the number of the buffer.
- {recursive} (boolean?) Apply callbacks recursively for all children. Any new children will also inherit the callbacks.

LanguageTree:source()

LanguageTree:source()

Returns the source content of the language tree (bufnr or string).

Return:

(integer|string)

LanguageTree:tree_for_range()

LanguageTree:tree_for_range({range}, {opts})

Gets the tree that contains {range}.

Parameters:

- {range} (table) A table with the following fields:
  - {[1]} (integer) start row

- {[2]} (integer) start column
- {[3]} (integer) end row
- {[4]} (integer) end column

- {opts} (table?) A table with the following fields:
  - {ignore_injections}? (boolean, default: true) Ignore injected languages

Return:
(TSTree?)

LanguageTree:trees() LanguageTree:trees()

Returns all trees of the regions parsed by this parser. Does not include child languages. The result is list-like if

- this LanguageTree is the root, in which case the result is empty or a singleton list; or
- the root LanguageTree is fully parsed.

Return:
(table<integer, TSTree>)

Lua module: vim.treesitter.query lua-treesitter-query

This Lua treesitter-query interface allows you to create queries and use them to parse text. See vim.treesitter.query.parse() for a working example.

TSQuery
Extends: userdata

Reference to an object held by the treesitter library that is used as a component of the vim.treesitter.Query for language feature support. See treesitter-query for more about queries or vim.treesitter.query.parse() for an example of how to obtain a query object.

Fields:
- {disable_capture} (fun(self: TSQuery, capture_name: string)) See TSQuery:disable_capture().
- {disable_pattern} (fun(self: TSQuery, pattern_index: integer)) See TSQuery:disable_pattern().

vim.treesitter.Query
Parsed query, see vim.treesitter.query.parse()

Fields:
- {captures} (string[]) list of (unique) capture names defined in query
- {has_combined_injections} (boolean) whether the query contains combined injections
- {has_conceal_line} (boolean) whether the query sets conceal_lines metadata
- {info} (vim.treesitter.QueryInfo) query context (e.g. captures, predicates, directives)
- {iter_captures} (fun(self: vim.treesitter.Query, node: TSNode, source: integer|string, start_row: integer?, end_row: integer?, opts: table?): fun(end_line: integer?, end_col: integer?): integer, TSNode, vim.treesitter.query.TSMetadata, TSQueryMatch, TSTree) See Query:iter_captures().
- {iter_matches} (fun(self: vim.treesitter.Query, node: TSNode, source: integer|string, start: integer?, stop: integer?, opts: table?): fun(): integer, table<integer, TSNode[]>, vim.treesitter.query.TSMetadata, TSTree) See Query:iter_matches().
- {lang} (string) parser language name

- {query} (TSQuery) userdata query object

vim.treesitter.query.add_directive()

add_directive({name}, {handler}, {opts})
    Adds a new directive to be used in queries

    Handlers can set match level data by setting directly on the metadata
    object metadata.key = value. Additionally, handlers can set node level
    data by using the capture id on the metadata table
    metadata[capture_id].key = value

Parameters:
- {name} (string) Name of the directive, without leading #

- {handler} (fun(match: table<integer,TSNode[]>, pattern: integer, source: integer|string, predicate: any[], metadata: vim.treesitter.query.TSMetadata))

    - match: A table mapping capture IDs to a list of captured nodes
    - pattern: the index of the matching pattern in the query
    - predicate: list of strings containing the full directive file being called, e.g. (node (#set! conceal "-")) would get the predicate { "#set!", "conceal", "-" }
- {opts} (table) A table with the following fields:

    - {force}? (boolean) Override an existing predicate of the same name

vim.treesitter.query.add_predicate()

add_predicate({name}, {handler}, {opts})
    Adds a new predicate to be used in queries

Parameters:
- {name} (string) Name of the predicate, without leading #

- {handler} (fun(match: table<integer,TSNode[]>, pattern: integer, source: integer|string, predicate: any[], metadata: vim.treesitter.query.TSMetadata): boolean?)

    - see vim.treesitter.query.add_directive() for argument meanings
- {opts} (table?) A table with the following fields:

    - {force}? (boolean) Override an existing predicate of the same name

edit({lang}) vim.treesitter.query.edit()

    Opens a live editor to query the buffer you started from.

    Can also be shown with the :EditQuery command. :EditQuery <tab>
    completes available parsers.

    If you move the cursor to a capture name ("@foo"), text matching the
    capture is highlighted with hl-DiagnosticVirtualTextHint in the source
    buffer. The query editor is a scratch buffer, use :write to save it. You
    can find example queries at $VIMRUNTIME/queries/.

Parameters:
- {lang} (string?) language to open the query editor for. If omitted, inferred from the current buffer's filetype.

get({lang}, {query_name}) vim.treesitter.query.get()

    Returns the runtime query {query_name} for {lang}.

Parameters:
- {lang} (string) Language to use for the query

- {query_name} (string) Name of the query (e.g. "highlights")

Return:
(vim.treesitter.Query?) Parsed query. nil if no query files are found.

vim.treesitter.query.get_files()

get_files({lang}, {query_name}, {is_included})
Gets the list of files used to make up a query

Parameters:
- {lang} (string) Language to get query for
- {query_name} (string) Name of the query to load (e.g., "highlights")
- {is_included} (boolean?) Internal parameter, most of the time left as nil

Return:
(string[]) query_files List of files to load for given query and language

lint({buf}, {opts}) vim.treesitter.query.lint()

Lint treesitter queries using installed parser, or clear lint errors.

Use treesitter-parsers in runtimepath to check the query file in {buf} for errors:
- verify that used nodes are valid identifiers in the grammar.
- verify that predicates and directives are valid.
- verify that top-level s-expressions are valid.

The found diagnostics are reported using diagnostic-api. By default, the parser used for verification is determined by the containing folder of the query file, e.g., if the path ends in /lua/highlights.scm, the parser for the lua language will be used.

Parameters:
- {buf} (integer) Buffer handle
- {opts} (table?) Optional keyword arguments:
  - {clear}? (boolean) Just clear current lint errors
  - {langs}? (string|string[]) Language(s) to use for checking the query. If multiple languages are specified, queries are validated for all of them

list_directives() vim.treesitter.query.list_directives()

Lists the currently available directives to use in queries.

Return:
(string[]) Supported directives.

list_predicates() vim.treesitter.query.list_predicates()

Lists the currently available predicates to use in queries.

Return:
(string[]) Supported predicates.

omnifunc({findstart}, {base}) vim.treesitter.query.omnifunc()

Omnifunc for completing node names and predicates in treesitter queries.

Use via

```
vim.bo.omnifunc = vim.treesitter.query.omnifunc
```

Parameters:
- {findstart} (0|1)
- {base} (string)

parse({lang}, {query}) vim.treesitter.query.parse()

Parses a {query} string and returns a Query object

(lua-treesitter-query), which can be used to search the tree for the query patterns (via Query:iter_captures(), Query:iter_matches()), or inspect/modify the query via these fields:

- captures: a list of unique capture names defined in the query (alias: info.captures).
- info.patterns: information about predicates.
- query: the underlying TSQuery which can be used to disable patterns or captures.

Example:

```
local query = vim.treesitter.query.parse('vimdoc', [[
  ; query
  ((h1) @str
    (#trim! @str 1 1 1 1))
]])
local tree = vim.treesitter.get_parser():parse()[1]
for id, node, metadata in query:iter_captures(tree:root(), 0) do
   -- Print the node name and source text.
   vim.print({node:type(), vim.treesitter.get_node_text(node, vim.api.nvim_get_current_buf())})
end
```

Parameters:

- {lang} (string) Language to use for the query
- {query} (string) Query text, in s-expr syntax

Return:

(vim.treesitter.Query) Parsed query

See also:

- vim.treesitter.query.get()

Query:iter_captures()

```
Query:iter_captures({node}, {source}, {start_row}, {end_row}, {opts})
```

Iterates over all captures from all matches in {node}.

{source} is required if the query contains predicates; then the caller must ensure to use a freshly parsed tree consistent with the current text of the buffer (if relevant). {start} and {stop} can be used to limit matches inside a row range (this is typically used with root node as the {node}, i.e., to get syntax highlight matches in the current viewport). When omitted, the {start} and {stop} row values are used from the given node.

The iterator returns four values:

1. the numeric id identifying the capture
2. the captured node
3. metadata from any directives processing the match
4. the match itself

Example: how to get captures by name:

```
for id, node, metadata, match in query:iter_captures(tree:root(), bufnr, first, last) do
  local name = query.captures[id] -- name of the capture in the query
  -- typically useful info about the node:
  local type = node:type() -- type of the captured node
  local row1, col1, row2, col2 = node:range() -- range of the capture
  -- ... use the info here ...
end
```

Note:

- Captures are only returned if the query pattern of a specific capture contained predicates.

Parameters:

- {node} (TSNode) under which the search will occur
- {source} (integer|string) Source buffer or string to extract

text from
- {start_row} (integer?) Starting line for the search. Defaults to

node:start().
- {end_row} (integer?) Stopping line for the search (end-inclusive,

unless stop_col is provided). Defaults to
node:end_().
- {opts} (table?) Optional keyword arguments:

  - end_col (integer) Stopping column for the search

(end-exclusive).
  - match_limit (integer) Set the maximum number of

in-progress matches (Default: none).
  - max_start_depth (integer) if non-zero, sets the maximum

start depth for each match. This is used to prevent
traversing too deep into a tree.
  - start_col (integer) Starting column for the search.

Return:
(fun(end_line: integer?, end_col: integer?): integer, TSNode, vim.treesitter.query.TSMetadata, TSQueryMatch, TSTree)
capture-id, capture-node, metadata, match, tree

Query:iter_matches()

Query:iter_matches({node}, {source}, {start}, {stop}, {opts})
Iterates the matches of self on a given range.

Iterate over all matches within a {node}. The arguments are the same as
for Query:iter_captures() but the iterated values are different: an
(1-based) index of the pattern in the query, a table mapping capture
indices to a list of nodes, and metadata from any directives processing
the match.

Example:

```
for pattern, match, metadata in cquery:iter_matches(tree:root(), bufnr, 0, -1) do
  for id, nodes in pairs(match) do
    local name = query.captures[id]
    for _, node in ipairs(nodes) do
      -- `node` was captured by the `name` capture in the match

      local node_data = metadata[id] -- Node level metadata
      -- ... use the info here ...
    end
  end
end
```

Parameters:
- {node} (TSNode) under which the search will occur

- {source} (integer|string) Source buffer or string to search

- {start} (integer?) Starting line for the search. Defaults to

node:start().
- {stop} (integer?) Stopping line for the search (end-exclusive).

Defaults to node:end_().
- {opts} (table?) Optional keyword arguments:

  - match_limit (integer) Set the maximum number of

in-progress matches (Default: none).
  - max_start_depth (integer) if non-zero, sets the maximum

start depth for each match. This is used to prevent
traversing too deep into a tree.

Return:
(fun(): integer, table<integer, TSNode[]>, vim.treesitter.query.TSMetadata, TSTree)
pattern-id, match, metadata, tree

set({lang}, {query_name}, {text}) vim.treesitter.query.set()

Sets the runtime query named {query_name} for {lang}

This allows users to override or extend any runtime files and/or configuration set by plugins.

For example, you could enable spellchecking of C identifiers with the following code:

```
vim.treesitter.query.set(
  'c',
  'highlights',
  [[;inherits c
  (identifier) @spell]]
)
```

Parameters:

- {lang} (string) Language to use for the query
- {query_name} (string) Name of the query (e.g., "highlights")
- {text} (string) Query text (unparsed).

TSQuery:disable_capture({capture_name}) TSQuery:disable_capture()

Disable a specific capture in this query; once disabled the capture cannot be re-enabled. {capture_name} should not include a leading "@".

Example: To disable the @variable.parameter capture from the vimdoc highlights query:

```
local query = vim.treesitter.query.get('vimdoc', 'highlights')
query.query:disable_capture("variable.parameter")
vim.treesitter.get_parser():parse()
```

Parameters:

- {capture_name} (string)

TSQuery:disable_pattern({pattern_index}) TSQuery:disable_pattern()

Disable a specific pattern in this query; once disabled the pattern cannot be re-enabled. The {pattern_index} for a particular match can be obtained with :Inspect!, or by reading the source of the query (i.e. from vim.treesitter.query.get_files()).

Example: To disable | links in vimdoc but keep other @markup.links highlighted:

```
local link_pattern = 9 -- from :Inspect!
local query = vim.treesitter.query.get('vimdoc', 'highlights')
query.query:disable_pattern(link_pattern)
local tree = vim.treesitter.get_parser():parse()[1]
```

Parameters:

- {pattern_index} (integer)

Terminal UI tui

By default when you run `nvim` (without --embed or --headless) it starts the builtin "terminal UI" (TUI). This default UI is optional: you can run Nvim as a "headless" server, or you can use a GUI.

Startup startup-tui startup-terminal

Nvim has a client-server architecture: by default when you run `nvim`, this starts the builtin UI client, which starts a `nvim --embed` server (child) process that the UI client connects to. After attaching to the server, the UI client calls nvim_set_client_info() (as recommended for all UIs dev-ui) and sets these fields on its channel:

```
client = {
  attributes = {
    license = 'Apache 2',
    pid = …,
    website = 'https://neovim.io',
  },
  name = 'nvim-tui',
  type = 'ui',
  version = { … },
},
-- Preserved across :restart.
detach = <false if the UI started the server, true after :detach!>,
```

Nvim guesses the terminal type when it starts (except in --embed and --headless modes). The $TERM environment variable is the primary hint that determines the terminal type.

terminfo E557 E558 E559 To display its user interface, Nvim reads a list of "terminal capabilities" from the system terminfo database (or builtin defaults if terminfo is not found). If that information is wrong, the screen may be messed up or keys may not be recognized.

The Unibilium library (used to read terminfo) allows you to override the system terminfo with one in the "$HOME/.terminfo/" directory. Building your own terminfo is usually as simple as running this:

```
curl -LO https://invisible-island.net/datafiles/current/terminfo.src.gz
gunzip terminfo.src.gz
tic -x terminfo.src
```

$NVIM_TERMDEFS

To override the builtin terminfo, set $NVIM_TERMDEFS to a JSON object whose keys correspond to one or more of the fields below, each of which will override its counterpart (if any) in the builtin terminfo. The system terminfo will be skipped entirely: only builtin definitions, plus the $NVIM_TERMDEFS overrides, will be used.

For example, to disable alternate screen, override the "reset the cursor style" sequence, and set the <Home> and <S-Home> sequences:

```
NVIM_TERMDEFS='{
  "enter_ca_mode": "",
  "exit_ca_mode": "",
  "reset_cursor_style": "\u001b[0 q",
  "key_home": ["\u001bOH", "\u001b[1;2H"]
}' nvim
```

Be careful when putting escape sequences in strings, as JSON uses unicode escapes like `\u001b`. For all available fields, see tui-termdefs.

$TERM

The $TERM environment variable must match the terminal you are using! Otherwise Nvim cannot know what sequences your terminal expects, and weird or sub-optimal behavior will result (scrolling quirks, wrong colors, etc.).

$TERM is also important because it is forwarded by SSH to the remote session, unlike most other environment variables.

```
For this terminal           Set $TERM to                 |builtin-terms|
-------------------------------------------------------------------------
ansi                        (unknown)                          Y
Ghostty                     ghostty, xterm-ghostty             Y
iTerm (original)            iterm, iTerm.app                   N
iTerm2 (new capabilities)   iterm2, iTerm2.app                 Y
Konsole                     konsole-256color                   N
libvte-based                vte, vte-256color                  Y
 (e.g. GNOME Terminal)      (aliases: gnome, gnome-256color)
Linux virtual terminal      linux, linux-256color              Y
PuTTY                       putty, putty-256color              Y
rxvt                        rxvt, rxvt-256color                Y
screen                      screen, screen-256color            Y
simple terminal (st)        st, st-256color                    Y
Terminal.app                nsterm                             N
tmux                        tmux, tmux-256color                Y
Windows/ConEmu              conemu                             Y
Windows/Cygwin-built Nvim   cygwin                             Y
Windows/Interix             interix                            Y
Windows/legacy console      win32con                           Y
Windows/VTP console         vtpcon                             Y
xterm or compatible         xterm, xterm-256color              Y
```

builtin-terms builtin_terms Nvim will map $TERM to a builtin entry according to the above table if:

- a terminfo database is not available on the system
- no terminfo entry found for the current detected terminal
- Nvim was built without terminfo support (`has('terminfo')==0`)

The builtin terminfo is not combined with any external terminfo database, nor can it be used in preference to one. You can thus entirely override any omissions or out-of-date information in the builtin terminfo database by supplying an external one with entries for the terminal type.

For example "TERM=putty-256color" will be mapped to the builtin "putty" entry. See also tui-colors.

Settings depending on terminal term-dependent-settings

If you want to set terminal-dependent options or mappings, you can do this in your init.vim. Example:

```
if $TERM =~ '^\(rxvt\|screen\|interix\|putty\)\(-.*\)\?$'
    set notermguicolors
elseif $TERM =~ '^\(tmux\|iterm\|vte\|gnome\)\(-.*\)\?$'
    set termguicolors
elseif $TERM =~ '^\(xterm\)\(-.*\)\?$'
    if $XTERM_VERSION != ''
        set termguicolors
    elseif $KONSOLE_PROFILE_NAME != ''
        set termguicolors
    elseif $VTE_VERSION != ''
        set termguicolors
    else
        set notermguicolors
    endif
elseif $TERM =~ ...
    ... and so forth ...
endif
```

scroll-region xterm-scroll-region Where possible, Nvim will use the terminal's ability to set a scroll region in order to redraw faster when a window is scrolled. If the terminal's terminfo description describes an ability to set top and bottom scroll margins, that is used.

This will not speed up scrolling in a window that is not the full width of the terminal. Xterm has an extra ability, not described by terminfo, to set left and right scroll margins as well. If Nvim detects that the terminal is Xterm, it will make use of this ability to speed up scrolling that is not the full width of the terminal.

tui-input

Historically, terminal emulators could not distinguish between certain control key modifiers and other keys. For example, <C-I> and <Tab> are represented in the same way, as are <Esc> and <C-[>, <CR> and <C-M>, and <NL> and <C-J>.

Modern terminal emulators are able to distinguish between these pairs of keys by encoding control modifiers differently. There are two common but distinct ways of doing this, known as "modifyOtherKeys" and "CSI u". Nvim supports both encoding methods and at startup will tell the terminal emulator that it understands these key encodings. If your terminal emulator supports it then this will allow you to map the key pairs listed above separately. <Tab>

Nvim uses libtermkey to convert terminal escape sequences to key codes. terminfo is used first, and CSI sequences not in terminfo (including extended keys a.k.a. "modifyOtherKeys" or "CSI u") can also be parsed.

For example, when running Nvim in tmux, this makes Nvim leave Insert mode and go to the window below:

```
tmux send-keys 'Escape' [ 2 7 u 'C-W' j
```

Where `'Escape' [ 2 7 u` is an unambiguous "CSI u" sequence for the <Esc> key.

The kitty keyboard protocol https://sw.kovidgoyal.net/kitty/keyboard-protocol/ is partially supported, including keypad keys in Unicode Private Use Area. For example, this sequence is recognized by Nvim as <C-kEnter>:

```
CSI 57414 ; 5 u
```

and can be used differently from <C-CR> in mappings.

tui-modifyOtherKeys tui-csiu At startup Nvim will query your terminal to see if it supports the "CSI u" encoding by writing the sequence

```
CSI ? u CSI c
```

If your terminal emulator responds with

```
CSI ? <flags> u
```

this means your terminal supports the "CSI u" encoding and Nvim will tell your terminal to enable it by writing the sequence

```
CSI > 3 u
```

If your terminal does not support "CSI u" then Nvim will instead enable the "modifyOtherKeys" encoding by writing the sequence

```
CSI > 4 ; 2 m
```

The "3" requests the kitty keyboard protocol "progressive enhancement" flags 1 (disambiguate escape codes) and 2 (report event types). Nvim does not request 8 (report all keys as escape codes) nor 16 (report associated text), which are needed to distinguish keys that produce the same text as their unmodified counterpart, such as <S-Space> and <Space>.

When Nvim exits cleanly it will send the corresponding sequence to disable the special key encoding. If Nvim does not exit cleanly then your terminal emulator could be in a bad state. If this happens, simply run "reset".

tui-colors

Nvim uses 256 colours by default, ignoring terminfo for most terminal types, including "linux" (whose virtual terminals have had 256-colour support since 4.8) and anything claiming to be "xterm". Also when $COLORTERM or $TERM contain the string "256".

Nvim similarly assumes that any terminal emulator that sets $COLORTERM to any value, is capable of at least 16-colour operation.

true-color xterm-true-color Nvim emits true (24-bit) colours in the terminal, if 'termguicolors' is set.

It uses the "setrgbf" and "setrgbb" terminfo extensions (proposed by Rüdiger Sonderfeld in 2013). If your terminfo definition is missing them, then Nvim will decide whether to add them to your terminfo definition, using the ISO 8613-6:1994/ITU T.416:1993 control sequences for setting RGB colours (but modified to use semicolons instead of colons unless the terminal is known to follow the standard).

Another convention, pioneered in 2016 by tmux, is the "Tc" terminfo extension. If terminfo has this flag, Nvim will add constructed "setrgbf" and "setrgbb" capabilities as if they had been in the terminfo definition.

If terminfo does not (yet) have this flag, Nvim will fall back to $TERM and other environment variables. It will add constructed "setrgbf" and "setrgbb" capabilities in the case of the "rxvt", "linux", "st", "tmux", and "iterm" terminal types, or when Konsole, genuine Xterm, a libvte terminal emulator version 0.36 or later, or a terminal emulator that sets the COLORTERM environment variable to "truecolor" is detected.

xterm-resize

Nvim can resize the terminal display on some terminals that implement an extension pioneered by dtterm. terminfo does not have a flag for this extension. So Nvim simply assumes that (all) "dtterm", "xterm", "teraterm", "rxvt" terminal types, and Konsole, are capable of this.

tui-cursor-shape

Nvim will adjust the shape of the cursor from a block to a line when in insert mode (or as specified by the 'guicursor' option), on terminals that support it. It uses the same terminfo extensions that were pioneered by tmux for this: "Ss" and "Se". Similarly, if you set the cursor highlight group with blend=100, Nvim hides the cursor through the "cvvis" and "civis" extensions.

If your terminfo definition is missing them, then Nvim will decide whether to add them to your terminfo definition, by looking at $TERM and other environment variables. For the "rxvt", "putty", "linux", "screen", "teraterm", and "iterm" terminal types, or when Konsole, a libvte-based terminal emulator, or genuine Xterm are detected, it will add constructed "Ss" and "Se" capabilities.

tui-cursor-tmux

Within tmux it may appear that Nvim is not changing the cursor, but in fact it is tmux receiving instructions from Nvim to change the cursor and not knowing what to do in turn. tmux must translate what it receives from Nvim into whatever control sequence is appropriate for the host terminal. It shares a common mechanism with Nvim, of using the "Ss" and "Se" capabilities from terminfo (for the output terminal) if they are present. Unlike Nvim, if they are not in terminfo you must add them by setting "terminal-overrides" in ~/.tmux.conf .

See the tmux(1) manual page for the details of how and what to do in the tmux configuration file. It will look something like:

```
set -ga terminal-overrides '*:Ss=\E[%p1%d q:Se=\E[ q'
```

or (alas!) for Konsole 18.07.70 or older, something more complex like:

```
set -ga terminal-overrides 'xterm*:\E]50;CursorShape=%?%p1%{3}%<%t%{0}%e%{1}%;%d\007'
```

Window size

window-size

[This is about the size of the whole window Vim is using, not a window that is created with the ":split" command.]

On Unix systems, three methods are tried to get the window size:

- an ioctl call (TIOCGSIZE or TIOCGWINSZ, depends on your system)
- the environment variables "LINES" and "COLUMNS"
- from the terminfo entries "lines" and "columns"

If everything fails a default size of 24 lines and 80 columns is assumed. If a window-resize signal is received the size will be set again. If the window size is wrong you can use the 'lines' and 'columns' options to set the correct values. See :mode.

Slow and fast terminals

slow-fast-terminal

slow-terminal

If you have a slow terminal you may want to reset the 'showcmd' and 'ruler' options. The command characters and cursor positions will not be shown in the status line (which involves a lot of cursor motions and attribute changes for every keypress or movement). If the terminal scrolls very slowly, set the 'scrolljump' to 5 or so. If the cursor is moved off the screen (e.g., with "j") Vim will scroll 5 lines at a time. Another possibility is to reduce the number of lines that Vim uses with the command "z{height}<CR>".

If the characters from the terminal are arriving with more than 1 second between them you might want to set the 'timeout' and/or 'ttimeout' option.

If you are using a color terminal that is slow when displaying lines beyond the end of a buffer, this is because Nvim is drawing the whitespace twice, in two sets of colours and attributes. To prevent this, use this command:

```
hi NonText cterm=NONE ctermfg=NONE
```

This draws the spaces with the default colours and attributes, which allows the second pass of drawing to be optimized away. **Note:** Although in theory the colours of whitespace are immaterial, in practice they change the colours of cursors and selections that cross them. This may have a visible, but minor, effect on some UIs.

Terminfo override

tui-termdefs

Below are the available fields in $NVIM_TERMDEFS. For keys of type `string[]`, the first entry is the unshifted variant of the key, and the second entry is the shifted variant.

```
Field                       Type
---------------------------------
RGB                         bool
Su                          bool
Tc                          bool
back_color_erase            bool

columns                     int
lines                       int
max_colors                  int

carriage_return             string
change_scroll_region        string
clear_screen                string
clr_eol                     string
clr_eos                     string
cursor_address              string
cursor_down                 string
cursor_home                 string
cursor_invisible            string
cursor_left                 string
cursor_normal               string
cursor_right                string
cursor_up                   string
delete_line                 string
enter_blink_mode            string
enter_bold_mode             string
enter_ca_mode               string
enter_dim_mode              string
enter_italics_mode          string
enter_reverse_mode          string
enter_secure_mode           string
enter_standout_mode         string
enter_underline_mode        string
erase_chars                 string
exit_attribute_mode         string
exit_ca_mode                string
from_status_line            string
insert_line                 string
keypad_local                string
keypad_xmit                 string
parm_delete_line            string
parm_down_cursor            string
parm_insert_line            string
parm_left_cursor            string
parm_right_cursor           string
parm_up_cursor              string
set_a_background            string
set_a_foreground            string
set_attributes              string
set_lr_margin               string
to_status_line              string

enter_strikethrough_mode    string
reset_cursor_color          string
reset_cursor_style          string
set_cursor_color            string
set_cursor_style            string
set_rgb_background          string
set_rgb_foreground          string
set_underline_style         string

key_backspace               string
key_btab                    string
key_clear                   string
key_npage                   string
key_ppage                   string
key_select                  string

key_beg                     string[]
key_dc                      string[]
key_end                     string[]
key_find                    string[]
key_home                    string[]
key_ic                      string[]
key_left                    string[]
key_right                   string[]
key_suspend                 string[]
key_undo                    string[]

key_f1                      string
...                         string
key_f63                     string
```

Uganda copying copyright license

SUMMARY

Kuwasha

Vim is Charityware. You can use and copy it as much as you like, but you are encouraged to make a donation for needy children in Uganda. Please see kcc below or visit the Kuwasha web site, available at the following URL:

https://www.kuwasha.net

You can also sponsor the development of Vim. Vim sponsors can vote for features. See sponsor. The money goes to Uganda anyway.

iccf ICCF

ICCF Holland and Kuwasha

Bram Moolenaar's charity, ICCF Holland, has long supported the education of children in Uganda through the Kibaale Children's Centre. Following Bram's passing in 2023, ICCF Holland transferred all activities to its sister charity Kuwasha in Canada and dissolved at the end of 2025.

Donations from Vim users are still welcome and will continue to go directly to Uganda. To continue supporting this cause, please send contributions to Kuwasha.

License

The Open Publication License applies to the Vim documentation, see manual-copyright.

=== begin of license ===

VIM LICENSE

I) There are no restrictions on distributing unmodified copies of Vim except that they must include this license text. You can also distribute unmodified parts of Vim, likewise unrestricted except that they must include this license text. You are also allowed to include executables that you made from the unmodified Vim sources, plus your own usage examples and Vim scripts.

II) It is allowed to distribute a modified (or extended) version of Vim, including executables and/or source code, when the following four conditions are met:
1) This license text must be included unmodified.
2) The modified Vim must be distributed in one of the following five ways:
a) If you make changes to Vim yourself, you must clearly describe in the distribution how to contact you. When the maintainer asks you (in any way) for a copy of the modified Vim you distributed, you must make your changes, including source code, available to the maintainer without fee. The maintainer reserves the right to include your changes in the official version of Vim. What the maintainer will do with your changes and under what license they will be distributed is negotiable. If there has been no negotiation then this license, or a later version, also applies to your changes. The current maintainers are listed here:
https://github.com/orgs/vim/people
If this changes it will be announced in appropriate places (most likely vim.sf.net, www.vim.org and/or comp.editors). When it is completely impossible to contact the maintainer, the obligation to send him your changes ceases. Once the maintainer has confirmed that he has received your changes they will not have to be sent again.
b) If you have received a modified Vim that was distributed as mentioned under a) you are allowed to further distribute it unmodified, as mentioned at I). If you make additional changes the text under a) applies to those changes.
c) Provide all the changes, including source code, with every copy of the modified Vim you distribute. This may be done in the form of a context diff. You can choose what license to use for new code you add. The changes and their license must not restrict others from making their own changes to the official version of Vim.
d) When you have a modified Vim which includes changes as mentioned under c), you can distribute it without the source code for the changes if the following three conditions are met:

- The license that applies to the changes permits you to distribute

  the changes to the Vim maintainer without fee or restriction, and permits the Vim maintainer to include the changes in the official version of Vim without fee or restriction.
- You keep the changes for at least three years after last

  distributing the corresponding modified Vim. When the maintainer or someone who you distributed the modified Vim to asks you (in any way) for the changes within this period, you must make them available to him.
- You clearly describe in the distribution how to contact you. This

  contact information must remain valid for at least three years after last distributing the corresponding modified Vim, or as long

as possible.
e) When the GNU General Public License (GPL) applies to the changes, you can distribute the modified Vim under the GNU GPL version 2 or any later version.
3) A message must be added, at least in the output of the ":version" command and in the intro screen, such that the user of the modified Vim is able to see that it was modified. When distributing as mentioned under 2)e) adding the message is only required for as far as this does not conflict with the license used for the changes.
4) The contact information as required under 2)a) and 2)d) must not be removed or changed, except that the person himself can make corrections.

III) If you distribute a modified version of Vim, you are encouraged to use the Vim license for your changes and make them available to the maintainer, including the source code. The preferred way to do this is by e-mail or by uploading the files to a server and e-mailing the URL. If the number of changes is small (e.g., a modified Makefile) e-mailing a context diff will do. The e-mail address to be used is <maintainer@vim.org>

IV) It is not allowed to remove this license from the distribution of the Vim sources, parts of it or from a modified version. You may use this license for previous Vim releases instead of the license that they came with, at your option.

=== end of license ===

Note:

- If you are happy with Vim, please express that by reading the rest of this file and consider helping needy children in Uganda.
- If you want to support further Vim development consider becoming a sponsor. The money goes to Uganda anyway.
- According to Richard Stallman the Vim license is GNU GPL compatible. A few minor changes have been made since he checked it, but that should not make a difference.
- If you link Vim with a library that goes under the GNU GPL, this limits further distribution to the GNU GPL. Also when you didn't actually change anything in Vim.
- Once a change is included that goes under the GNU GPL, this forces all further changes to also be made under the GNU GPL or a compatible license.

Kibaale Children's Centre

kcc Kibaale charity

Kibaale Children's Centre (KCC) is located in Kibaale, a small town in the south of Uganda, near Tanzania, in East Africa. The area is known as Rakai District. The population is mostly farmers. Although people are poor, there usually is enough food. But this district is suffering from AIDS more than any other part of the world. Some say that it started there. Estimations are that in the past 10 to 30% of the Ugandans are infected with HIV. Because parents die, there are many orphans. In this district about 60,000 children have lost one or both parents, out of a population of 350,000. Although AIDS is now mostly under control, the problems are still continuing.

The children need a lot of help. The KCC is working hard to provide the needy with food, medical care and education. Food and medical care to keep them healthy now, and education so that they can take care of themselves in the future. KCC works on a Christian base, but help is given to children of any religion.

The key to solving the problems in this area is education. This has been neglected in the past years with president Idi Amin and the following civil wars. Now that the government is stable again, the children and parents have to learn how to take care of themselves and how to avoid infections. There is also help for people who are ill and hungry, but the primary goal is to prevent people from getting ill and to teach them how to grow healthy food.

Most of the orphans are living in an extended family. An uncle or older sister is taking care of them. Because these families are big and the income (if any) is low, a child is lucky if it gets healthy food. Clothes, medical care and schooling is beyond its reach. To help these needy children, a sponsorship program was put into place. A child can be financially adopted. For a few dollars a month KCC sees to it that the child gets indispensable items, is healthy, goes to school and KCC takes care of anything else that needs to be done for the child and the family that supports it.

Besides helping the child directly, the environment where the child grows up needs to be improved. KCC helps schools to improve their teaching methods. There is a demonstration school at the centre and teacher trainings are given. Health workers are being trained, hygiene education is carried out and households are stimulated to build a proper latrine. I helped setting up a production site for cement slabs. These are used to build a good latrine. They are sold below cost price.

There is a clinic at the project, which provides children and their family medical help. Since 2020 a maternity ward was added and 24/7 service is available. When needed, transport to a hospital is offered. Immunization programs are carried out and help is provided when an epidemic is breaking out (measles and cholera have been a problem).

Summer 1994 to summer 1995 Bram spent a whole year at the centre, working as a volunteer. Bram helped to expand the centre and worked in the area of water and sanitation. Bram learned that the help that the KCC provides really helps. When Bram came back to Holland, he wanted to continue supporting KCC. To do this he has been raising funds and organizing the sponsorship program.

How do you know that the money will be spent right? First of all you have the personal guarantee of Bram as the author of Vim, who knew the people working at the centre personally. Furthermore, the centre has been co-sponsored and inspected by World Vision, Save the Children Fund and is now under the supervision of Pacific Academy Outreach Society. Bram has visited the centre many times, starting in 1993. The visit reports are have been shared on the ICCF web site (may no longer be available).

If you have any further questions, send an e-mail: info@kuwasha.net.

The address of the centre is:

Kibaale Children's Centre
p.o. box 1658
Masaka, Uganda, East Africa

donate

Sending money:

Check the Kuwasha web site for the latest information!

Look on their site for information about sponsorship:
https://www.kuwasha.net/
If you make a donation to Kuwasha you will receive a tax receipt which can be submitted with your tax return.

Credit Card: You can use PayPal to send money with a Credit card. This is the most widely used Internet based payment system. It's really simple to use. Use this link to find more info:
https://www.paypal.com/ncp/payment/ZZPJ7XEYD3CZ2
The e-mail address for sending the money to is:
info@kuwasha.net

Undo and redo

The basics are explained in section 02.5 of the user manual.

1. Undo and redo commands undo-commands

<Undo> or undo <Undo> u
u Undo [count] changes. The cursor returns to its original position.

:u :un :undo
:u[ndo] Undo one change.

E830

:u[ndo] {N} Jump to after change number {N}. See undo-branches for the meaning of {N}.

:u[ndo]! Undo one change and remove it from undo history.

E5767

:u[ndo]! {N} Like ":u[ndo] {N}", but forget all changes in the current undo branch up until {N}. You may only use ":undo! {N}" to move backwards in the same undo branch, not to redo or switch to a different undo branch.

CTRL-R

CTRL-R Redo [count] changes which were undone.

:red :redo redo
:red[o] Redo one change which was undone.

U

U Undo all latest changes on one line, the line where the latest change was made. U itself also counts as a change, and thus U undoes a previous U.

The last changes are remembered. You can use the undo and redo commands above to revert the text to how it was before each change. You can also apply the changes again, getting back the text before the undo.

The "U" command is treated by undo/redo just like any other command. Thus a "u" command undoes a "U" command and a 'CTRL-R' command redoes it again. When mixing "U", "u" and 'CTRL-R' you will notice that the "U" command will restore the situation of a line to before the previous "U" command. This may be confusing. Try it out to get used to it.
The "U" command will always mark the buffer as changed. When "U" changes the buffer back to how it was without changes, it is still considered changed. Use "u" to undo changes until the buffer becomes unchanged.

2. Two ways of undo undo-two-ways

How undo and redo commands work depends on the 'u' flag in 'cpoptions'. There is the Vim way ('u' excluded) and the Vi-compatible way ('u' included). In the Vim way, "uu" undoes two changes. In the Vi-compatible way, "uu" does nothing (undoes an undo).

'u' excluded, the Vim way:
You can go back in time with the undo command. You can then go forward again with the redo command. If you make a new change after the undo command, the redo will not be possible anymore.

'u' included, the Vi-compatible way:
The undo command undoes the previous change, and also the previous undo command. The redo command repeats the previous undo command. It does NOT repeat a change command, use "." for that.

Examples	Vim way	Vi-compatible way
"uu"	two times undo	no-op
"u CTRL-R"	no-op	two times undo

Rationale: Nvi uses the "." command instead of CTRL-R. Unfortunately, this is not Vi compatible. For example "dwdwu." in Vi deletes two words, in Nvi it does nothing.

3. Undo blocks undo-blocks

One undo command normally undoes a typed command, no matter how many changes that command makes. This sequence of undo-able changes forms an undo block. Thus if the typed key(s) call a function, all the commands in the function are undone together.

If you want to write a function or script that doesn't create a new undoable change but joins in with the previous change use this command:

:undoj :undojoin E790
:undoj[oin] Join further changes with the previous undo block.
Warning: Use with care, it may prevent the user from properly undoing changes. Don't use this after undo or redo.

This is most useful when you need to prompt the user halfway through a change. For example in a function that calls getchar(). Do make sure that there was

a related change before this that you must join with.

This doesn't work by itself, because the next key press will start a new change again. But you can do something like this:

```
:undojoin | delete
```

After this a "u" command will undo the delete command and the previous change.

undo-break undo-close-block

To do the opposite, use a new undo block for the next change, in Insert mode use CTRL-G u. This is useful if you want an insert command to be undoable in parts. E.g., for each sentence. i_CTRL-G_u

Setting the value of 'undolevels' also closes the undo block. Even when the new value is equal to the old value. Use g:undolevels to explicitly read and write only the global value of 'undolevels'.

```
let &g:undolevels = &g:undolevels
```

Note that the similar-looking assignment let &undolevels=&undolevels does not preserve the global option value of 'undolevels' in the event that the local option has been set to a different value. For example:

```
" Start with different global and local values for 'undolevels'.
let &g:undolevels = 1000
let &l:undolevels = 2000
" This assignment changes the global option to 2000:
let &undolevels = &undolevels
```

4. Undo branches

undo-branches undo-tree

Above we only discussed one line of undo/redo. But it is also possible to branch off. This happens when you undo a few changes and then make a new change. The undone changes become a branch. You can go to that branch with the following commands. The undo-tree of a file can be visualized and interactively applied using :Undotree.

This is explained in the user manual: usr_32.txt.

:undol :undolist

:undol[ist] List the leafs in the tree of changes. Example:

```
number changes  when               saved
       88      88  2010/01/04 14:25:53
      108     107  08/07 12:47:51
      136      46  13:33:01             7
      166     164  3 seconds ago
```

The "number" column is the change number. This number continuously increases and can be used to identify a specific undo-able change, see :undo.
The "changes" column is the number of changes to this leaf from the root of the tree.
The "when" column is the date and time when this change was made. The four possible formats are:

```
    N seconds ago
    HH:MM:SS             hour, minute, seconds
    MM/DD HH:MM:SS       idem, with month and day
    YYYY/MM/DD HH:MM:SS  idem, with year
```

The "saved" column specifies, if this change was written to disk and which file write it was. This can be used with the :later and :earlier commands.
For more details use the undotree() function.

g-

g- Go to older text state. With a count repeat that many times.

:ea :earlier

:ea[rlier] {count} Go to older text state {count} times.
:ea[rlier] {N}s Go to older text state about {N} seconds before.
:ea[rlier] {N}m Go to older text state about {N} minutes before.
:ea[rlier] {N}h Go to older text state about {N} hours before.
:ea[rlier] {N}d Go to older text state about {N} days before.

:ea[rlier] {N}f Go to older text state {N} file writes before.
When changes were made since the last write ":earlier 1f" will revert the text to the state when it was written. Otherwise it will go to the write before that.
When at the state of the first file write, or when the file was not written, ":earlier 1f" will go to before the first change.

g+

g+ Go to newer text state. With a count repeat that many times.

:lat :later

:lat[er] {count} Go to newer text state {count} times.

:lat[er] {N}s		Go to newer text state about {N} seconds later.
:lat[er] {N}m		Go to newer text state about {N} minutes later.
:lat[er] {N}h		Go to newer text state about {N} hours later.
:lat[er] {N}d		Go to newer text state about {N} days later.

:lat[er] {N}f		Go to newer text state {N} file writes later.
			When at the state of the last file write, ":later 1f"
			will go to the newest text state.

Note that text states will become unreachable when undo information is cleared for 'undolevels'.

Don't be surprised when moving through time shows multiple changes to take place at a time. This happens when moving through the undo tree and then making a new change.

EXAMPLE

Start with this text:
one two three

Delete the first word by pressing "x" three times:
ne two three
e two three
two three

Now undo that by pressing "u" three times:
e two three
ne two three
one two three

Delete the second word by pressing "x" three times:
one wo three
one o three
one three

Now undo that by using "g-" three times:
one o three
one wo three
two three

You are now back in the first undo branch, after deleting "one". Repeating "g-" will now bring you back to the original text:
e two three
ne two three
one two three

Jump to the last change with ":later 1h":
one three

And back to the start again with ":earlier 1h":
one two three

Note that using "u" and CTRL-R will not get you to all possible text states while repeating "g-" and "g+" does.

5. Undo persistence

undo-persistence persistent-undo

When unloading a buffer Vim normally destroys the tree of undos created for that buffer. By setting the 'undofile' option, Vim will automatically save your undo history when you write a file and restore undo history when you edit the file again.

The 'undofile' option is checked after writing a file, before the BufWritePost autocommands. If you want to control what files to write undo information for, you can use a BufWritePre autocommand:

```
au BufWritePre /tmp/* setlocal noundofile
```

Vim saves undo trees in a separate undo file, one for each edited file, using a simple scheme that maps filesystem paths directly to undo files. Vim will detect if an undo file is no longer synchronized with the file it was written for (with a hash of the file contents) and ignore it when the file was changed after the undo file was written, to prevent corruption. An undo file is also ignored if its owner differs from the owner of the edited file, except when the owner of the undo file is the current user. Set 'verbose' to get a message about that when opening a file.

Location of the undo files is controlled by the 'undodir' option, by default they are saved to the dedicated directory in the application data folder.

You can also save and restore undo histories by using ":wundo" and ":rundo" respectively:

:wu :wundo
:wu[ndo][!] {file}
		Write undo history to {file}.
		When {file} exists and it does not look like an undo file
		(the magic number at the start of the file is wrong), then
		this fails, unless the ! was added.
		If it exists and does look like an undo file it is
		overwritten. If there is no undo-history, nothing will be
		written.
		Implementation detail: Overwriting happens by first deleting
		the existing file and then creating a new file with the same
		name. So it is not possible to overwrite an existing undofile
		in a write-protected directory.

:rund :rundo

:rund[o] {file} Read undo history from {file}.

You can use these in autocommands to explicitly specify the name of the history file. E.g.:

```
au BufReadPost * call ReadUndo()
au BufWritePost * call WriteUndo()
func ReadUndo()
  if filereadable(expand('%:h') .. '/UNDO/' .. expand('%:t'))
    rundo %:h/UNDO/%:t
  endif
endfunc
func WriteUndo()
  let dirname = expand('%:h') .. '/UNDO'
  if !isdirectory(dirname)
    call mkdir(dirname)
  endif
  wundo %:h/UNDO/%:t
endfunc
```

You should keep 'undofile' off, otherwise you end up with two undo files for every write.

You can use the undofile() function to find out the file name that Vim would use.

Note that while reading/writing files and 'undofile' is set most errors will be silent, unless 'verbose' is set. With :wundo and :rundo you will get more error messages, e.g., when the file cannot be read or written.

NOTE: undo files are never deleted by Vim. You need to delete them yourself.

Reading an existing undo file may fail for several reasons:
E822 It cannot be opened, because the file permissions don't allow it.
E823 The magic number at the start of the file doesn't match. This usually means it is not an undo file.
E824 The version number of the undo file indicates that it's written by a newer version of Vim. You need that newer version to open it. Don't write the buffer if you want to keep the undo info in the file.
"File contents changed, cannot use undo info"
The file text differs from when the undo file was written. This means the undo file cannot be used, it would corrupt the text. This also happens when 'encoding' differs from when the undo file was written.
E825 The undo file does not contain valid contents and cannot be used.
"Not reading undo file, owner differs"
The undo file is owned by someone else than the owner of the text file. For safety the undo file is not used.

Writing an undo file may fail for these reasons:
E828 The file to be written cannot be created. Perhaps you do not have write permissions in the directory.
"Cannot write undo file in any directory in 'undodir'"
None of the directories in 'undodir' can be used.
"Will not overwrite with undo file, cannot read"
A file exists with the name of the undo file to be written, but it cannot be read. You may want to delete this file or rename it.
"Will not overwrite, this is not an undo file"
A file exists with the name of the undo file to be written, but it does not start with the right magic number. You may want to delete this file or rename it.
"Skipping undo file write, nothing to undo"
There is no undo information to be written, nothing has been changed or 'undolevels' is negative.
E829 An error occurred while writing the undo file. You may want to try again.

6. Remarks about undo undo-remarks

The number of changes that are remembered is set with the 'undolevels' option. If it is zero, the Vi-compatible way is always used. If it is negative no undo is possible. Use this if you are running out of memory.

clear-undo

When you set 'undolevels' to -1 the undo information is not immediately cleared, this happens at the next change. To force clearing the undo information you can use these commands:

```
:let old_undolevels = &l:undolevels
:setlocal undolevels=-1
:exe "normal a \<BS>\<Esc>"
:let &l:undolevels = old_undolevels
:unlet old_undolevels
```

Note use of &l:undolevels to explicitly read the local value of 'undolevels' and the use of :setlocal to change only the local option (which takes precedence over the corresponding global option value). Saving the option value via the use of &undolevels is unpredictable; it reads either the local value (if one has been set) or the global value (otherwise). Also, if a local value has been set, changing the option via :set undolevels will change both

the global and local values, requiring extra work to save and restore both values.

Marks for the buffer ('a to 'z) are also saved and restored, together with the text.

When all changes have been undone, the buffer is not considered to be changed. It is then possible to exit Vim with ":q" instead of ":q!".
Note that this is relative to the last write of the file. Typing "u" after ":w" actually changes the buffer, compared to what was written, so the buffer is considered changed then.

When manual folding is being used, the folds are not saved and restored. Only changes completely within a fold will keep the fold as it was, because the first and last line of the fold don't change.

The numbered registers can also be used for undoing deletes. Each time you delete text, it is put into register "1. The contents of register "1 are shifted to "2, etc. The contents of register "9 are lost. You can now get back the most recent deleted text with the put command: '"1P'. (also, if the deleted text was the result of the last delete or copy operation, 'P' or 'p' also works as this puts the contents of the unnamed register). You can get back the text of three deletes ago with '"3P'.

redo-register

If you want to get back more than one part of deleted text, you can use a special feature of the repeat command ".". It will increase the number of the register used. So if you first do '"1P', the following "." will result in a '"2P'. Repeating this will result in all numbered registers being inserted.

Example:	If you deleted text with 'dd....' it can be restored with
		'"1P....'.

If you don't know in which register the deleted text is, you can use the :display command. An alternative is to try the first register with '"1P', and if it is not what you want do 'u.'. This will remove the contents of the first put, and repeat the put command for the second register. Repeat the 'u.' until you got what you want.

Defining and using functions.

This is introduced in section 41.7 of the user manual.

1. Defining a function ~

New functions can be defined. These can be called just like builtin functions. The function executes a sequence of Ex commands. Normal mode commands can be executed with the :normal command.

The function name must start with an uppercase letter, to avoid confusion with builtin functions. To prevent from using the same name in different scripts make them script-local. If you do use a global function then avoid obvious, short names. A good habit is to start the function name with the name of the script, e.g., "HTMLcolor()".

It is also possible to use curly braces, see curly-braces-names.

The autoload facility is useful to define a function only when it's called.

local-function

A function local to a script must start with "s:". A local script function can only be called from within the script and from functions, user commands and autocommands defined in the script. It is also possible to call the function from a mapping defined in the script, but then <SID> must be used instead of "s:" when the mapping is expanded outside of the script.
There are only script-local functions, no buffer-local or window-local functions.

:fu :function E128 E129 E123 E454 E1058 E1068

:fu[nction] List all functions and their arguments.

:fu[nction][!] {name} List function {name}, annotated with line numbers unless "!" is given.
{name} may be a Dictionary Funcref entry:

```
:function dict.init
```

Note that {name} is not an expression, you cannot use a variable that is a function reference. You can use this dirty trick to list the function referred to with variable "Funcref":

```
let g:MyFuncref = Funcref
func g:MyFuncref
unlet g:MyFuncref
```

:fu[nction] /{pattern} List functions with a name matching {pattern}.
Example that lists all functions ending with "File":

```
:function /File$
```

:function-verbose

When 'verbose' is non-zero, listing a function will also display where it was last defined. Example:

```
:verbose function SetFileTypeSH
    function SetFileTypeSH(name)
        Last set from /usr/share/vim/vim-7.0/filetype.vim
```

See :verbose-cmd for more information.

E124 E125 E853 E884

:fu[nction][!] {name}([arguments]) [range] [abort] [dict] [closure]
Define a new function by the name {name}. The body of the function follows in the next lines, until the matching :endfunction.

The name must be made of alphanumeric characters and '_' and must start with a capital or "s:" (see above). Note that using "b:", "l:", etc. is not allowed (since patch 7.4.260 E884 is given if the function name has a colon, e.g. for "foo:bar()"), while a leading "g:" is skipped and still requires a following capital letter.

{name} may be a Dictionary Funcref entry:

```
:function dict.init(arg)
```

"dict" must be an existing dictionary. The entry "init" is added if it didn't exist yet. Otherwise [!] is required to overwrite an existing function. The result is a Funcref to a numbered function. The function can only be used with a Funcref and will be deleted if there are no more references to it.

E127 E122

When a function by this name already exists and [!] is not used an error message is given. There is one exception: When sourcing a script again, a function that was previously defined in that script will be silently replaced.
When [!] is used, an existing function is silently

replaced. Unless it is currently being executed, that is an error.
NOTE: Use ! wisely. If used without care it can cause an existing function to be replaced unexpectedly, which is hard to debug.

For the {arguments} see function-argument.

:func-range a:firstline a:lastline
When the [range] argument is added, the function is expected to take care of a range itself. The range is passed as "a:firstline" and "a:lastline". If [range] is excluded, ":{range}call" will call the function for each line in the range, with the cursor on the start of each line. See function-range-example.
The cursor is still moved to the first line of the range, as is the case with all Ex commands.

:func-abort
When the [abort] argument is added, the function will abort as soon as an error is detected.

:func-dict
When the [dict] argument is added, the function must be invoked through an entry in a Dictionary. The local variable "self" will then be set to the dictionary. See Dictionary-function.

:func-closure E932
When the [closure] argument is added, the function can access variables and arguments from the outer scope. This is usually called a closure. In this example Bar() uses "x" from the scope of Foo(). It remains referenced even after Foo() returns:

```
:function! Foo()
:  let x = 0
:  function! Bar() closure
:    let x += 1
:    return x
:  endfunction
:  return funcref('Bar')
:endfunction

:let F = Foo()
:echo F()
```

```
1
```

```
:echo F()
```

```
2
```

```
:echo F()
```

```
3
```

function-search-undo
The last used search pattern and the redo command "." will not be changed by the function. This also implies that the effect of :nohlsearch is undone when the function returns.

:endf :endfunction E126 E193 W22
:endf[unction] [argument]
The end of a function definition. Best is to put it on a line by its own, without [argument].

[argument] can be:
```
        | command       command to execute next
        \n command      command to execute next
        " comment       always ignored
        anything else   ignored, warning given when
                        'verbose' is non-zero
```
The support for a following command was added in Vim 8.0.0654, before that any argument was silently ignored.

To be able to define a function inside an :execute command, use line breaks instead of :bar:

```
:exe "func Foo()\necho 'foo'\nendfunc"
```

:delf :delfunction E130 E131 E933
:delf[unction][!] {name}
Delete function {name}.
{name} can also be a Dictionary entry that is a Funcref:

```
:delfunc dict.init
```

This will remove the "init" entry from "dict". The function is deleted if there are no more references to

it.
With the ! there is no error if the function does not exist.

:retu :return E133

:retu[rn] [expr] Return from a function. When [expr] is given, it is evaluated and returned as the result of the function. If [expr] is not given, the number 0 is returned. When a function ends without an explicit ":return", the number 0 is returned.
Note that there is no check for unreachable lines, thus there is no warning if commands follow ":return".
Also, there is no check if the following line contains a valid command. Forgetting the line continuation backslash may go unnoticed:

```
return 'some text'
       .. ' some more text'
```

Will happily return "some text" without an error. It should have been:

```
return 'some text'
       \ .. ' some more text'
```

If the ":return" is used after a :try but before the matching :finally (if present), the commands following the ":finally" up to the matching :endtry are executed first. This process applies to all nested ":try"s inside the function. The function returns at the outermost ":endtry".

function-argument a:var

An argument can be defined by giving its name. In the function this can then be used as "a:name" ("a:" for argument).

a:0 a:1 a:000 E740 ...

Up to 20 arguments can be given, separated by commas. After the named arguments an argument "..." can be specified, which means that more arguments may optionally be following. In the function the extra arguments can be used as "a:1", "a:2", etc. "a:0" is set to the number of extra arguments (which can be 0). "a:000" is set to a List that contains these arguments. Note that "a:1" is the same as "a:000[0]".

E742

The a: scope and the variables in it cannot be changed, they are fixed. However, if a composite type is used, such as List or Dictionary , you can change their contents. Thus you can pass a List to a function and have the function add an item to it. If you want to make sure the function cannot change a List or Dictionary use :lockvar.

It is also possible to define a function without any arguments. You must still supply the () then.

It is allowed to define another function inside a function body.

optional-function-argument

You can provide default values for positional named arguments. This makes them optional for function calls. When a positional argument is not specified at a call, the default expression is used to initialize it. This only works for functions declared with :function, not for lambda expressions expr-lambda.

Example:

```
function Something(key, value = 10)
   echo a:key .. ": " .. a:value
endfunction
call Something('empty') "empty: 10"
call Something('key', 20)       "key: 20"
```

The argument default expressions are evaluated at the time of the function call, not when the function is defined. Thus it is possible to use an expression which is invalid the moment the function is defined. The expressions are also only evaluated when arguments are not specified during a call.

E989

Optional arguments with default expressions must occur after any mandatory arguments. You can use "..." after all optional named arguments.

It is possible for later argument defaults to refer to prior arguments, but not the other way around. They must be prefixed with "a:", as with all arguments.

Example that works:

```
:function Okay(mandatory, optional = a:mandatory)
:endfunction
```

Example that does NOT work:

```
:function NoGood(first = a:second, second = 10)
:endfunction
```

When not using "...", the number of arguments in a function call must be at least equal to the number of mandatory named arguments. When using "...", the number of arguments may be larger than the total of mandatory and optional arguments.

local-variables

Inside a function local variables can be used. These will disappear when the function returns. Global variables need to be accessed with "g:". Inside functions local variables are accessed without prepending anything. But you can also prepend "l:" if you like. This is required for some reserved names, such as "version".

Example:

```
:function Table(title, ...)
:  echohl Title
:  echo a:title
:  echohl None
:  echo a:0 .. " items:"
:  for s in a:000
:    echon ' ' .. s
:  endfor
:endfunction
```

This function can then be called with:

```
call Table("Table", "line1", "line2")
call Table("Empty Table")
```

To return more than one value, return a List:

```
:function Compute(n1, n2)
:  if a:n2 == 0
:    return ["fail", 0]
:  endif
:  return ["ok", a:n1 / a:n2]
:endfunction
```

This function can then be called with:

```
:let [success, div] = Compute(102, 6)
:if success == "ok"
:  echo div
:endif
```

1. Calling a function ~

:cal :call E107 E117

:[range]cal[l] {name}([arguments])
Call a function. The name of the function and its arguments are as specified with :function. Up to 20 arguments can be used. The returned value is discarded.
Without a range and for functions that accept a range, the function is called once. When a range is given the cursor is positioned at the start of the first line before executing the function.
When a range is given and the function doesn't handle it itself, the function is executed for each line in the range, with the cursor in the first column of that line. The cursor is left at the last line (possibly moved by the last function call). The arguments are re-evaluated for each line. Thus this works:

function-range-example

```
:function Mynumber(arg)
:  echo line(".") .. " " .. a:arg
:endfunction
:1,5call Mynumber(getline("."))
```

The "a:firstline" and "a:lastline" are defined anyway, they can be used to do something different at the start or end of the range.

Example of a function that handles the range itself:

```
:function Cont() range
:  execute (a:firstline + 1) .. "," .. a:lastline .. 's/^/\t\\ '
:endfunction
:4,8call Cont()
```

This function inserts the continuation character "\" in front of all the lines in the range, except the first one.

When the function returns a composite value it can be further
dereferenced, but the range will not be used then. Example:

```
:4,8call GetDict().method()
```

Here GetDict() gets the range but method() does not.

E132

The recursiveness of user functions is restricted with the 'maxfuncdepth'
option.

It is also possible to use :eval. It does not support a range, but does
allow for method chaining, e.g.:

```
eval GetList()->Filter()->append('$')
```

A function can also be called as part of evaluating an expression or when it
is used as a method:

```
let x = GetList()
let y = GetList()->Filter()
```

1. Cleaning up in a function ~

:defe :defer
:defe[r] {func}({args}) Call {func} when the current function is done.
{args} are evaluated here.

Quite often a command in a function has a global effect, which must be undone
when the function finishes. Handling this in all kinds of situations can be a
hassle. Especially when an unexpected error is encountered. This can be done
with try / finally blocks, but this gets complicated when there is more
than one.

A much simpler solution is using defer. It schedules a function call when
the function is returning, no matter if there is an error. Example:

```
func Filter(text) abort
  call writefile(a:text, 'Tempfile')
  call system('filter < Tempfile > Outfile')
  call Handle('Outfile')
  call delete('Tempfile')
  call delete('Outfile')
endfunc
```

Here 'Tempfile' and 'Outfile' will not be deleted if something causes the
function to abort. :defer can be used to avoid that:

```
func Filter(text) abort
  call writefile(a:text, 'Tempfile')
  defer delete('Tempfile')
  defer delete('Outfile')
  call system('filter < Tempfile > Outfile')
  call Handle('Outfile')
endfunc
```

Note that deleting "Outfile" is scheduled before calling system(), since it
can be created even when system() fails.

The deferred functions are called in reverse order, the last one added is
executed first. A useless example:

```
func Useless() abort
  for s in range(3)
    defer execute('echomsg "number ' .. s .. '"')
  endfor
endfunc
```

Now :messages shows:
        number 2
        number 1
        number 0

Any return value of the deferred function is discarded. The function cannot
be followed by anything, such as "->func" or ".member". Currently
:defer GetArg()->TheFunc() does not work, it may work in a later version.

Errors are reported but do not cause aborting execution of deferred functions
or altering execution outside of deferred functions.

No range is accepted. The function can be a partial with extra arguments, but
not with a dictionary. E1300

1. Automatically loading functions ~

autoload-functions

When using many or large functions, it's possible to automatically define them

only when they are used. There are two methods: with an autocommand and with the "autoload" directory in 'runtimepath'.

Using an autocommand

This is introduced in the user manual, section 41.14.

The autocommand is useful if you have a plugin that is a long Vim script file. You can define the autocommand and quickly quit the script with :finish. That makes Vim startup faster. The autocommand should then load the same file again, setting a variable to skip the :finish command.

Use the FuncUndefined autocommand event with a pattern that matches the function(s) to be defined. Example:

```
:au FuncUndefined BufNet* source ~/vim/bufnetfuncs.vim
```

The file "~/vim/bufnetfuncs.vim" should then define functions that start with "BufNet". Also see FuncUndefined.

Using an autoload script

autoload E746

This is introduced in the user manual, section 41.15.

Using a script in the "autoload" directory is simpler, but requires using exactly the right file name. A function that can be autoloaded has a name like this:

```
:call filename#funcname()
```

When such a function is called, and it is not defined yet, Vim will search the "autoload" directories in 'runtimepath' for a script file called "filename.vim". For example "~/.config/nvim/autoload/filename.vim". That file should then define the function like this:

```
function filename#funcname()
   echo "Done!"
endfunction
```

If the file doesn't exist, Vim will also search in 'packpath' (under "start") to allow calling packages' functions from your vimrc when the packages have not been added to 'runtimepath' yet (see packages).

The file name and the name used before the # in the function must match exactly, and the defined function must have the name exactly as it will be called.

It is possible to use subdirectories. Every # in the function name works like a path separator. Thus when calling a function:

```
:call foo#bar#func()
```

Vim will look for the file "autoload/foo/bar.vim" in 'runtimepath'.

This also works when reading a variable that has not been set yet:

```
:let l = foo#bar#lvar
```

However, when the autoload script was already loaded it won't be loaded again for an unknown variable.

When assigning a value to such a variable nothing special happens. This can be used to pass settings to the autoload script before it's loaded:

```
:let foo#bar#toggle = 1
:call foo#bar#func()
```

Note that when you make a mistake and call a function that is supposed to be defined in an autoload script, but the script doesn't actually define the function, you will get an error message for the missing function. If you fix the autoload script it won't be automatically loaded again. Either restart Vim or manually source the script.

Also note that if you have two script files, and one calls a function in the other and vice versa, before the used function is defined, it won't work. Avoid using the autoload functionality at the toplevel.

Hint: If you distribute a bunch of scripts read distribute-script.

1. Various commands

various-cmds

CTRL-C

CTRL-C Interrupt the current command, script, mapping, prompt, or search.

In Normal mode, any pending command is aborted. During a search, interrupts the search.

Lua code cannot be interrupted unless it "yields" via vim.wait().

CTRL-L

CTRL-L Clears and redraws the screen. The redraw may happen later, after processing typeahead.
See also nvim__redraw().

CTRL-L-default

By default, also clears search highlighting :nohlsearch, removes multicursors, and updates diffs :diffupdate. default-mappings

:mod :mode E359

:mod[e] Clears and redraws the screen.
See also nvim__redraw().

:redr :redraw

:redr[aw][!] Redraws pending screen updates now, or the entire screen if "!" is included. To CLEAR the screen use :mode or CTRL-L.
It can be used to redraw the screen in a script or function (or a mapping if 'lazyredraw' is set).
See also nvim__redraw().

:redraws :redrawstatus

:redraws[tatus][!] Redraws the status line and window bar of the current window, or all status lines and window bars if "!" is included. Redraws the commandline instead if it contains the 'ruler'. Useful if 'statusline' or 'winbar' includes an item that doesn't cause automatic updating.
See also nvim__redraw().

:redrawt :redrawtabline

:redrawt[abline] Redraw the tabline. Useful to update the tabline when 'tabline' includes an item that doesn't trigger automatic updating. See also nvim__redraw().

N<Del>

<Del> When entering a number: Remove the last digit.
Note: if you like to use <BS> for this, add this mapping to your vimrc:

```
:map CTRL-V <BS>   CTRL-V <Del>
```

ga :as :ascii

:as[cii] or
ga Print the ascii value of the character under the cursor in decimal, hexadecimal and octal.
Mnemonic: Get Ascii value.

For example, when the cursor is on a 'R':

<R> 82, Hex 52, Octal 122

When the character is a non-standard ASCII character, the non-printable version is also given.

When the character is larger than 127:

<<81>> 129, Hex 0081, Octal 201
<p> 254, Hex 00fe, Octal 376
(where <p> is a special character)

The <Nul> character in a file is stored internally as <NL>, but it will be shown as:

<^@> 0, Hex 00, Octal 000

If the character has composing characters these are also shown. The value of 'maxcombine' doesn't matter.

If the character can be inserted as a digraph, also output the two characters that can be used to create the character:

<ö> 246, Hex 00f6, Oct 366, Digr o:
This shows you can type CTRL-K o : to insert ö.

g8

g8 Print the hex values of the bytes used in the character under the cursor, assuming it is in UTF-8 encoding. This also shows composing characters. The value of 'maxcombine' doesn't matter.
Example of a character with two composing characters:

e0 b8 81 + e0 b8 b9 + e0 b9 89

8g8

8g8 Find an illegal UTF-8 byte sequence at or after the cursor.
Can be used when editing a file that was supposed to be UTF-8 but was read as if it is an 8-bit encoding because it contains illegal bytes.
Does not wrap around the end of the file.
Note that when the cursor is on an illegal byte or the cursor is halfway through a multibyte character the command won't move the cursor.

gx

gx Opens the current filepath, URL (decided by <cfile>, 'isfname'), or LSP "documentLink" at cursor using the system default handler.

In help files, gx treats help tags as https://neovim.io/doc/user/ URLs.

Mapped to vim.ui.open().

v_gx

{Visual}gx Opens the selected text using the system default handler. Mapped to vim.ui.open().

:p :pr :print E749

:[range]p[rint] [flags]
Print [range] lines (default current line). Can also be spelled :[range]| due to Vi compatibility (see :bar). Gives an error in an empty buffer.
In the GUI you can use the File.Print menu entry.
See ex-flags for [flags].
The :filter command can be used to only show lines matching a pattern.

:[range]p[rint] {count} [flags]
Print {count} lines, starting with [range] (default current line cmdline-ranges).
See ex-flags for [flags].

:l :list

:[range]l[ist] [count] [flags]
Same as :print, but show tabs as ">", trailing spaces as "-", and non-breakable space characters as "+" by default. Further changed by the 'listchars' option.
See ex-flags for [flags].

:nu :number

:[range]nu[mber] [count] [flags]
Same as :print, but precede each line with its line number. (See also hl-LineNr and 'numberwidth').
See ex-flags for [flags].

:#

:[range]# [count] [flags]
synonym for :number.

:#! vim-shebang

:#!{anything} Ignored, so that you can start a Vim script with:

```
#!vim -S
let mylogbook='$HOME/logbook.md'
exe $':e {mylogbook}'
$
put ='## ' .. strftime('%d. %b %Y')
norm! o
```

Make that script executable and run it to create a new diary entry.

:z E144

:[range]z[+-^.=][count] Display several lines of text surrounding the line specified with [range], or around the current line if there is no [range].

If there is a [count], that's how many lines you'll see; if there is no [count] and only one window then twice the value of the 'scroll' option is used, otherwise the current window height minus 3 is used.
This is the value of "scr" in the table below.

If there is a [count] the 'window' option is set to its value.

:z can be used either alone or followed by any of several marks. These have the following effect:

mark	first line	last line	new cursor line
+	current line	1 scr forward	1 scr forward
-	1 scr back	current line	current line
^	2 scr back	1 scr back	1 scr back
.	1/2 scr back	1/2 scr fwd	1/2 scr fwd
=	1/2 scr back	1/2 scr fwd	current line

Specifying no mark at all is the same as "+".
If the mark is "=", a line of dashes is printed
around the current line.

If the 'number' option is set, absolute line numbers
will be included in the output.

:z!

:[range]z![+-^.=][count]
Like ":z", but when [count] is not specified, it
defaults to the Vim window height minus one.

:[range]z[!]#[+-^.=][count] :z#

Like ":z" or ":z!", but number the lines.

:=

:= [args] Without [args]: prints the last line number.
With [args]: equivalent to :lua ={expr}. see :lua

:{range}= Prints the last line number in {range}. For example,
this prints the current line number:

```
:.=
```

:norm[al][!] {commands} :norm :normal
Execute Normal mode commands {commands}. This makes
it possible to execute Normal mode commands typed on
the command-line. {commands} are executed like they
are typed. For undo all commands are undone together.
Execution stops when an error is encountered.

If the [!] is given, mappings will not be used.
Without it, when this command is called from a
non-remappable mapping (:noremap), the argument can
be mapped anyway.

{commands} should be a complete command. If
{commands} does not finish a command, the last one
will be aborted as if <Esc> or <C-C> was typed.
This implies that an insert command must be completed
(to start Insert mode, see :startinsert). A ":"
command must be completed as well. And you can't
start Ex-mode.

The display is not updated while ":normal" is busy.

{commands} cannot start with a space. Put a count of
1 (one) before it, "1 " is one space.

This command cannot be followed by another command,
since any '|' is considered part of the command.

This command can be used recursively, but the depth is
limited by 'maxmapdepth'.

An alternative is to use :execute, which uses an
expression as argument. This allows the use of
printable characters to represent special characters.

Example:

```
:exe "normal \<c-w>\<c-w>"
```

:{range}norm[al][!] {commands} :normal-range

Execute Normal mode commands {commands} for each line
in the {range}. Before executing the {commands}, the
cursor is positioned in the first column of the range,
for each line. Otherwise it's the same as the
":normal" command without a range.

:sh :shell E371 E360
:sh[ell] Removed. vim-differences

:terminal :te
:te[rminal][!] [cmd] Run [cmd] in a non-interactive 'shell' in a new
terminal-emulator buffer. Without [cmd], start an
interactive 'shell'.

By default the current window is used. To open in
a split window, use :horizontal or :vertical:

```
:hor te
:vert te
```

Type i to enter Terminal-mode, then keys are sent to
the job running in the terminal. Type <C-\><C-N> to
leave Terminal-mode. CTRL-_CTRL-N. Type <C-\><C-O>
to execute a single normal mode command t_CTRL-_CTRL-O

Fails if changes have been made to the current buffer,
unless 'hidden' is set.

If [cmd] is omitted, and the 'shell' job exits with no
error, the buffer is closed automatically
default-autocmds.

To enter Terminal-mode automatically:

```
autocmd TermOpen * startinsert
```

:!cmd :!

:!{cmd} Execute {cmd} with 'shell'. See also :terminal.
For the filter command, see :range!.

The command runs in a non-interactive shell connected to a pipe (not a terminal). Use :terminal to run an interactive shell connected to a terminal.

Backgrounded ("&") commands must not write to stdout or stderr, the streams are closed immediately. E5677
Use jobstart() instead.

```
:call jobstart('foo', {'detach':1})
```

For powershell, chaining a stringed executable path requires using the call operator (&).

```
:!Write-Output "1`n2" | & "C:\Windows\System32\sort.exe" /r
```

Vim builds command line using options 'shell', 'shcf', 'sxq' and 'shq' in the following order:
&sh &shcf &sxq &shq {cmd} &shq &sxq
So setting both 'sxq' and 'shq' is possible but rarely useful. Additional escaping inside {cmd} may also be due to 'sxe' option.

Also, all cmdline-special characters in {cmd} are replaced by Vim before passing them to shell.

E34

Any "!" in {cmd} is replaced with the previous external command (see also 'cpoptions'), unless escaped by a backslash. Example: ":!ls" followed by ":!echo ! \! \\!" executes "echo ls ! \!".

Any "|" in {cmd} is passed to the shell, you cannot use it to append a Vim command. See :bar.

Any "%" in {cmd} is expanded to the current file name.
Any "#" in {cmd} is expanded to the alternate file name.
Special characters are not escaped, use quotes or shellescape():

```
:!ls "%"
:exe "!ls " .. shellescape(expand("%"),1)
```

Newline character ends {cmd} unless a backslash precedes the newline. What follows is interpreted as another : command.

After the command has been executed, the timestamp and size of the current file is checked timestamp.

If the command produces too much output some lines may be skipped so the command can execute quickly. No data is lost, this only affects the display. The last few lines are always displayed (never skipped).

To avoid the hit-enter prompt use:

```
:silent !{cmd}
```

:!!

:!! Repeat last ":!{cmd}".

:ve :ver :version

:ve[rsion] Print editor version and build information.
See also feature-compile.

:redi :redir

:redi[r][!] > {file} Redirect messages to file {file}. The messages which are the output of commands are written to that file, until redirection ends. The messages are also still shown on the screen. When [!] is included, an existing file is overwritten. When [!] is omitted, and {file} exists, this command fails.

Only one ":redir" can be active at a time. Calls to ":redir" will close any active redirection before starting redirection to the new target. For recursive use check out execute().

To stop the messages and commands from being echoed to the screen, put the commands in a function and call it with ":silent call Function()".
Alternatives are the 'verbosefile' option or execute() function, these can be used in combination with ":redir".

:redi[r] >> {file} Redirect messages to file {file}. Append if {file} already exists.

:redi[r] @{a-zA-Z}
:redi[r] @{a-zA-Z}> Redirect messages to register {a-z}. Append to the
                     contents of the register if its name is given
                     uppercase {A-Z}. The ">" after the register name is
                     optional.
:redi[r] @{a-z}>>   Append messages to register {a-z}.

:redi[r] @*>
:redi[r] @+>   Redirect messages to the selection or clipboard. For
                     backward compatibility, the ">" after the register
                     name can be omitted. See quotestar and quoteplus.
:redi[r] @*>>
:redi[r] @+>>   Append messages to the selection or clipboard.

:redi[r] @">   Redirect messages to the unnamed register. For
                     backward compatibility, the ">" after the register
                     name can be omitted.
:redi[r] @">>   Append messages to the unnamed register.

:redi[r] => {var}   Redirect messages to a variable. If the variable
                     doesn't exist, then it is created. If the variable
                     exists, then it is initialized to an empty string.
                     The variable will remain empty until redirection ends.
                     Only string variables can be used. After the
                     redirection starts, if the variable is removed or
                     locked or the variable type is changed, then further
                     command output messages will cause errors. When using
                     a local variable (l:var in a function or s:var in a
                     script) and another :redir causes the current one to
                     end, the scope might be different and the assignment
                     fails.
                     To get the output of one command the execute()
                     function can be used instead of redirection.

:redi[r] =>> {var}   Append messages to an existing variable. Only string
                     variables can be used.

:redi[r] END   End redirecting messages.

                                                  :filt :filter
:filt[er][!] {pattern} {command}
:filt[er][!] /{pattern}/ {command}
                     Restrict the output of {command} to lines matching
                     with {pattern}. For example, to list only xml files:

```
:filter /\.xml$/ oldfiles
```

                     If the [!] is given, restrict the output of {command}
                     to lines that do NOT match {pattern}.

                     {pattern} is a Vim search pattern. Instead of
                     enclosing it in / any non-ID character (see 'isident')
                     can be used, so long as it does not appear in
                     {pattern}.
                     Without the enclosing character the pattern cannot
                     include the bar character. 'ignorecase' is not used.

                     The pattern is matched against the relevant part of
                     the output, not necessarily the whole line. Only some
                     commands support filtering, try it out to check if it
                     works. Some of the commands that support filtering:
                        :#   - filter whole line
                        :clist   - filter by file name or module name
                        :command   - filter by command name
                        :files   - filter by file name
                        :highlight - filter by highlight group
                        :history   - filter by history commands
                        :jumps   - filter by file name
                        :let   - filter by variable name
                        :list   - filter whole line
                        :llist   - filter by file name or module name
                        :marks   - filter by text in the current file,
                                      or file name for other files
                        :oldfiles   - filter by file name
                        :registers - filter by register contents
                                      (does not work multi-line)
                        :set   - filter by option name

                     Only normal messages are filtered, error messages are
                     not.

                                          :sil :silent :silent!
:sil[ent][!] {command} Execute {command} silently. Normal messages will not
                     be given or added to the message history.
                     When [!] is added, error messages will also be
                     skipped, and commands and mappings will not be aborted
                     when an error is detected. v:errmsg is still set.
                     When [!] is not used, an error message will cause
                     further messages to be displayed normally.
                     Redirection, started with :redir, will continue as
                     usual, although there might be small differences.
                     This will allow redirecting the output of a command
                     without seeing it on the screen. Example:

```
:redir >/tmp/foobar
:silent g/Aap/p
:redir END
```

To execute a Normal mode command silently, use the :normal command. For example, to search for a string without messages:

```
:silent exe "normal /path\<CR>"
```

":silent!" is useful to execute a command that may fail, but the failure is to be ignored. Example:

```
:let v:errmsg = ""
:silent! /^begin
:if v:errmsg != ""
: ... pattern was not found
```

":silent" also skips the hit-enter prompt. Dialogs that prompt for user input (confirm(), 'swapfile', …) are never silent.

:uns :unsilent

:uns[ilent] {command} Execute {command} not silently. Only makes a difference when :silent was used to get to this command.
Use this for giving a message even when :silent was used. In this example :silent is used to avoid the message about reading the file and :unsilent to be able to list the first line of each file.

```
:silent argdo unsilent echo expand('%') .. ": " .. getline(1)
```

:verb :verbose

:[count]verb[ose] {command}
Execute {command} with 'verbose' set to [count]. If [count] is omitted one is used. ":0verbose" can be used to set 'verbose' to zero.
The additional use of ":silent" makes messages generated but not displayed.
The combination of ":silent" and ":verbose" can be used to generate messages and check them with v:statusmsg and friends. For example:

```
:let v:statusmsg = ""
:silent verbose runtime foobar.vim
:if v:statusmsg != ""
:  " foobar.vim could not be found
:endif
```

When concatenating another command, the ":verbose" only applies to the first one:

```
:4verbose set verbose | set verbose
```

verbose=4
verbose=0

For logging verbose messages in a file use the 'verbosefile' option.

:verbose-cmd

When 'verbose' is non-zero, listing the value of a Vim option or a key map or an abbreviation or a user-defined function or a command or a highlight group or an autocommand will also display where it was last defined. If they were defined in Lua they will only be located if 'verbose' is set. So Start nvim with -V1 arg to see them. If it was defined manually then there will be no "Last set" message. When it was defined while executing a function, user command or autocommand, the script in which it was defined is reported.

K

[count]K Runs the program given by 'keywordprg' to lookup the word (defined by 'iskeyword') under or right of the cursor. Default is "man". Works like this:

```
:tabnew | terminal {program} {keyword}
```

Special cases:

- If 'keywordprg' begins with ":" it is invoked as

  a Vim command and [count] is passed as the first argument, if present.
- If 'keywordprg' is empty, :help is used.
- When 'keywordprg' is equal to "man", a [count]

  before "K" is inserted after the "man" command and

before the keyword. For example, using "2K" while
the cursor is on "mkdir", results in:

```
!man 2 mkdir
```

- When 'keywordprg' is equal to "man -s", a [count]

  before "K" is inserted after the "-s". If there is
  no count, the "-s" is removed.

K-lsp-default

- The Nvim LSP client sets K to show LSP "hover"

  feature. lsp-defaults

v_K

{Visual}K Like "K", but use the visually highlighted text for
the keyword. Only works when the highlighted text is
not more than one line.

gO

gO Show a filetype-specific, navigable "outline" of the
current buffer. For example, in a help buffer this
shows the table of contents.

Works in help and :Man buffers, or any buffer with
an active LSP client (lsp-defaults).

[N]gs gs :sl :sleep
:[N]sl[eep] [N][m] Do nothing for [N] seconds, or [N] milliseconds if [m]
was given. "gs" always uses seconds.
Default is one second.

```
:sleep       "sleep for one second
:5sleep      "sleep for five seconds
:sleep 100m     "sleep for 100 milliseconds
10gs         "sleep for ten seconds
```

Can be interrupted with CTRL-C.
"gs" stands for "goto sleep".
While sleeping the cursor is positioned in the text,
if at a visible position.
Queued messages are processed during the sleep.

:sl! :sleep!
:[N]sl[eep]! [N][m] Same as above, but hide the cursor.

g==

g== Executes the current code block.

Works in help buffers.

:upt :uptime
:upt[ime] Shows Nvim's uptime.
See also v:starttime.

2. Using Vim like less or more less

If you use the less or more program to view a file, you don't get syntax
highlighting. Thus you would like to use Vim instead. You can do this by
using the shell script "$VIMRUNTIME/scripts/less.sh".

This shell script uses the Vim script "$VIMRUNTIME/scripts/less.vim". It sets
up mappings to simulate the commands that less supports. Otherwise, you can
still use the Vim commands.

This isn't perfect. For example, when viewing a short file Vim will still use
the whole screen. But it works well enough for most uses, and you get syntax
highlighting.

The "h" key will give you a short overview of the available commands.

If you want to set options differently when using less, define the
LessInitFunc in your vimrc, for example:

```
func LessInitFunc()
  set nocursorcolumn nocursorline
endfunc
```

3. Commenting commenting

Nvim supports commenting and uncommenting of lines based on 'commentstring'.

Acting on a single line behaves as follows:

- If the line matches 'commentstring', the comment markers are removed (e.g.

  /*foo*/ is transformed to foo).
- Otherwise the comment markers are added to the current line (e.g. foo is

transformed to /*foo*/). Blank lines are ignored.

Acting on multiple lines behaves as follows:

- If each affected non-blank line matches 'commentstring', then all comment markers are removed.
- Otherwise all affected lines are converted to comments; blank lines are transformed to empty comments (e.g. /**/). Comment markers are aligned to the least indented line.

Matching 'commentstring' does not account for whitespace in comment markers. Removing comment markers is first attempted exactly, with fallback to using markers trimmed from whitespace.

If the filetype of the buffer is associated with a language for which a treesitter parser is installed, then vim.filetype.get_option() is called to look up the value of 'commentstring' corresponding to the cursor position. (This can be different from the buffer's 'commentstring' in case of treesitter-language-injections.)

The following default-mappings are defined:

		gc gc-default
gc{motion}	Comment or uncomment lines covered by {motion}.	
		gcc gcc-default
gcc	Comment or uncomment [count] lines starting at cursor.	
		v_gc v_gc-default
{Visual}gc	Comment or uncomment the selected line(s).	
		o_gc o_gc-default
gc	Text object for the largest contiguous block of non-blank commented lines around the cursor (e.g. gcgc uncomments a comment block; dgc deletes it). Works only in Operator-pending mode.	

Differences between Vim and Vi

1. Limits limits

Vim has only a few limits for the files that can be edited. Vi cannot handle <Nul> characters and characters above 128, has limited line length, many other limits.

Maximum line length	2147483647 characters
Maximum number of lines	2147483647 lines
Maximum file size	2147483647 bytes (2 Gbyte) when a long integer is 32 bits. Much more for 64 bit longs. Also limited by available disk space for the swap-file.

E75

Length of a file path	Unix and Win32: 1024 characters, otherwise 256 characters (or as much as the system supports).
Length of an expanded string option	Unix and Win32: 1024 characters, otherwise 256 characters
Maximum display width	Unix and Win32: 1024 characters, otherwise 255 characters
Maximum lhs of a mapping	50 characters.

Number of different highlighting types: over 30000
Range of a Number variable: -2147483648 to 2147483647 (might be more on 64 bit systems) See also: v:numbermax, v:numbermin and v:numbersize
Maximum length of a line in a tags file: 512 bytes.

E1541

Maximum value for /\U and /\%U: 2147483647 (for 32bit integer).

Information for undo and text in registers is kept in memory, thus when making (big) changes the amount of (virtual) memory available limits the number of undo levels and the text that can be kept in registers. Other things are also kept in memory: Command-line history, error messages for Quickfix mode, etc.

2. The most interesting additions vim-additions

Support for different systems.
Vim can be used on:

- Modern Unix systems (BSD, Linux, etc.)
- Windows (XP SP 2 or greater)
- OS X

Multi level persistent undo. undo
'u' goes backward in time, 'CTRL-R' goes forward again. Set option 'undolevels' to the number of changes to be remembered (default 1000). Set 'undolevels' to 0 for a Vi-compatible one level undo. Set it to -1 for no undo at all.
When all changes in a buffer have been undone, the buffer is not considered changed anymore. You can exit it with :q, without <!>.
When undoing a few changes and then making a new change Vim will create a branch in the undo tree. This means you can go back to any state of the text, there is no risk of a change causing text to be lost forever. undo-tree
The undo information is stored in a file when the 'undofile' option is set. This means you can exit Vim, start Vim on a previously edited file and undo changes that were made before exiting Vim.

Graphical User Interface (GUI). gui
Included support for GUI: menu's, mouse, scrollbars, etc. You can define your own menus. Better support for CTRL/SHIFT/ALT keys in combination with special keys and mouse. Supported for various platforms such as Win32.

Multiple windows and buffers. windows.txt
Vim can split the screen into several windows, each editing a different buffer or the same buffer at a different location. Buffers can still be loaded (and changed) but not displayed in a window. This is called a hidden buffer. Many commands and options have been added for this facility.
Vim can also use multiple tabpages, each with one or more windows. A line with tab labels can be used to quickly switch between these pages.
tabpage

Syntax highlighting. :syntax
Vim can highlight keywords, patterns and other things. This is defined by a number of :syntax commands, and can be made to highlight most languages and file types. A number of files are included for highlighting the most common languages, like C, C++, Java, Pascal, Makefiles, shell scripts, etc. The colors used for highlighting can be defined for ordinary terminals, color terminals and the GUI with the :highlight command. A convenient way to do this is using a :colorscheme command.

The highlighted text can be exported as HTML. convert-to-HTML
Other items that can be highlighted are matches with the search string 'hlsearch', matching parens matchparen and the cursor line and column 'cursorline' 'cursorcolumn'.

Spell checking. spell
When the 'spell' option is set Vim will highlight spelling mistakes. About 50 languages are currently supported, selected with the 'spelllang' option. In source code only comments and strings are checked for spelling.

Folding. folding
A range of lines can be shown as one "folded" line. This allows overviewing a file and moving blocks of text around quickly. Folds can be created manually, from the syntax of the file, by indent, etc.

Diff mode. diff-mode
Vim can show two versions of a file with the differences highlighted. Parts of the text that are equal are folded away. Commands can be used to move text from one version to the other.

Plugins. add-plugin
The functionality can be extended by dropping a plugin file in the right directory. That's an easy way to start using Vim scripts written by others. Plugins can be for all kind of files, or specifically for a filetype.
Packages make this even easier. packages

Asynchronous communication and timers. job-control timer
Vim can exchange messages with other processes in the background. Vim can start a job, communicate with it and stop it. job-control
Timers can fire once or repeatedly and invoke a function to do any work. timer

Repeat a series of commands. g
"q{c}" starts recording typed characters into named register {c}. A subsequent "q" stops recording. The register can then be executed with the "@{c}" command. This is very useful to repeat a complex action.

Flexible insert mode. ins-special-special
The arrow keys can be used in insert mode to move around in the file. This breaks the insert in two parts as far as undo and redo is concerned.

CTRL-O can be used to execute a single Normal mode command. This is almost the same as hitting <Esc>, typing the command and doing a.

Visual mode. Visual-mode
Visual mode can be used to first highlight a piece of text and then give a command to do something with it. This is an (easy to use) alternative to first giving the operator and then moving to the end of the text to be operated upon.
v and V are used to start Visual mode. v works on characters and V on lines. Move the cursor to extend the Visual area. It is shown highlighted on the screen. By typing "o" the other end of the Visual area can be moved. The Visual area can be affected by an operator:

d	delete
c	change
y	yank
> or <	insert or delete indent
!	filter through external program
=	filter through indent
:	start : command for the Visual lines.
gq	format text to 'textwidth' columns
J	join lines
~	swap case
u	make lowercase
U	make uppercase

Block operators. visual-block
With Visual mode a rectangular block of text can be selected. Start Visual mode with CTRL-V. The block can be deleted ("d"), yanked ("y") or its case can be changed ("~", "u" and "U"). A deleted or yanked block can be put into the text with the "p" and "P" commands.

Help system. :help
Help is displayed in a window. The usual commands can be used to move around, search for a string, etc. Tags can be used to jump around in the help files, just like hypertext links. The :help command takes an argument to quickly jump to the info on a subject. <F1> is the quick access to the help system. The name of the help index file can be set with the 'helpfile' option.

Command-line editing and history. cmdline-editing
You can insert or delete at any place in the command-line using the cursor keys. The right/left cursor keys can be used to move forward/backward one character. The shifted right/left cursor keys can be used to move forward/backward one word. CTRL-B/CTRL-E can be used to go to the begin/end of the command-line.
(Vi: can only alter the last character in the line)
(Vi: when hitting <Esc> the command-line is executed. This is unexpected for most people; therefore it was changed in Vim. But when the <Esc> is part of a mapping, the command-line is executed. If you want the Vi behaviour also when typing <Esc>, use ":cmap ^V<Esc>

^V^M")

cmdline-history

The command-lines are remembered. The up/down cursor keys can be used to recall previous command-lines. The 'history' option can be set to the number of lines that will be remembered. There is a separate history for commands and for search patterns.

Command-line completion. cmdline-completion

While entering a command-line (on the bottom line of the screen) <Tab> can be typed to complete

what **example**

- command :e<Tab>
- tag :ta scr<Tab>
- option :set sc<Tab>
- option value :set hf=<Tab>
- file name :e ve<Tab>
- etc.

If there are multiple matches, CTRL-N (next) and CTRL-P (previous) will walk through the matches. <Tab> works like CTRL-N, but wraps around to the first match.

The 'wildchar' option can be set to the character for command-line completion, <Tab> is the default. CTRL-D can be typed after an (incomplete) wildcard; all matches will be listed. CTRL-A will insert all matches. CTRL-L will insert the longest common part of the matches.

Insert-mode completion. ins-completion

In Insert mode the CTRL-N and CTRL-P keys can be used to complete a word that appears elsewhere. i_CTRL-N

With CTRL-X another mode is entered, through which completion can be done for:

i_CTRL-X_CTRL-F	file names
i_CTRL-X_CTRL-K	words from 'dictionary' files
i_CTRL-X_CTRL-T	words from 'thesaurus' files
i_CTRL-X_CTRL-I	words from included files
i_CTRL-X_CTRL-L	whole lines
i_CTRL-X_CTRL-]	words from the tags file
i_CTRL-X_CTRL-D	definitions or macros
i_CTRL-X_CTRL-O	Omni completion: clever completion specifically for a file type
i_CTRL-X_CTRL-R	contents from registers

etc.

Long line support. 'wrap' 'linebreak'

If the 'wrap' option is off, long lines will not wrap and only part of them will be shown. When the cursor is moved to a part that is not shown, the screen will scroll horizontally. The minimum number of columns to scroll can be set with the 'sidescroll' option. The zh and zl commands can be used to scroll sideways.
Alternatively, long lines are broken in between words when the 'linebreak' option is set. This allows editing a single-line paragraph conveniently (e.g. when the text is later read into a DTP program). Move the cursor up/down with the gk and gj commands.

Text formatting. formatting

The 'textwidth' option can be used to automatically limit the line length. This supplements the 'wrapmargin' option of Vi, which was not very useful. The gq operator can be used to format a piece of text (for example, ggqap formats the current paragraph). Commands for text alignment: :center, :left and :right.

Extended search patterns. pattern

There are many extra items to match various text items. Examples:
A "\n" can be used in a search pattern to match a line break.
"x\{2,4}" matches "x" 2 to 4 times.
"\s" matches a white space character.

Directory, remote and archive browsing. dir

Vim can browse the file system. Simply edit a directory. Move around in the list with the usual commands and press <Enter> to go to the directory or file under the cursor.
This also works for remote files over ftp, http, ssh, etc.
Zip and tar archives can also be browsed. tar zip

Edit-compile-edit speedup. quickfix

The :make command can be used to run the compilation and jump to the first error. A file with compiler error messages is interpreted. Vim jumps to the first error.

Each line in the error file is scanned for the name of a file, line number and error message. The 'errorformat' option can be set to a

list of scanf-like strings to handle output from many compilers.

The :cn command can be used to jump to the next error.
:cl lists all the error messages. Other commands are available.
The 'makeef' option has the name of the file with error messages.
The 'makeprg' option contains the name of the program to be executed
with the :make command.
The 'shellpipe' option contains the string to be used to put the
output of the compiler into the errorfile.

Finding matches in files. :vimgrep
Vim can search for a pattern in multiple files. This uses the
advanced Vim regexp pattern, works on all systems and also works to
search in compressed files.

Improved indenting for programs. 'cindent'
When the 'cindent' option is on the indent of each line is
automatically adjusted. C syntax is mostly recognized. The indent
for various styles can be set with 'cinoptions'. The keys to trigger
indenting can be set with 'cinkeys'.

Comments can be automatically formatted. The 'comments' option can be
set to the characters that start and end a comment. This works best
for C code, but also works for e-mail (">" at start of the line) and
other types of text. The = operator can be used to re-indent
lines.

For many other languages an indent plugin is present to support
automatic indenting. 30.3

Searching for words in included files. include-search
The [i command can be used to search for a match of the word under
the cursor in the current and included files. The 'include' option
can be set to a pattern that describes a command to include a file
(the default is for C programs).
The [I command lists all matches, the [_CTRL-I command jumps to
a match.
The [d, [D and [_CTRL-D commands do the same, but only for
lines where the pattern given with the 'define' option matches.

Automatic commands. autocommand
Commands can be automatically executed when reading a file, writing a
file, jumping to another buffer, etc., depending on the file name.
This is useful to set options and mappings for C programs,
documentation, plain text, e-mail, etc. This also makes it possible
to edit compressed files.

Scripts and Expressions. expression
Commands have been added to form up a powerful script language.
:if   Conditional execution, which can be used for example
                to set options depending on the value of $TERM.
:while   Repeat a number of commands.
:for   Loop over a list.
:echo   Print the result of an expression.
:let   Assign a value to an internal variable, option, etc.
                Variable types are Number, String, List and Dictionary.
:execute   Execute a command formed by an expression.
:try   Catch exceptions.
etc., etc. See eval.
Debugging and profiling are supported. debug-scripts profile
If this is not enough, an interface is provided to Python.

Viminfo.
The command-line history, marks and registers can be stored in a file
that is read on startup. This can be used to repeat a search command
or command-line command after exiting and restarting Vim. It is also
possible to jump right back to where the last edit stopped with '0.
The 'viminfo' option can be set to select which items to store in the
.viminfo file. This is off by default.

Mouse support. mouse-using
The mouse is supported in the GUI version, in an xterm for Unix, for
BSDs with sysmouse, for Linux with gpm, and for Win32. It can be used
to position the cursor, select the visual area, paste a register, etc.

Usage of key names. <> key-notation
Special keys now all have a name like <Up>, <End>, etc.
This name can be used in mappings, to make it easy to edit them.

Editing binary files. edit-binary
Vim can edit binary files. You can change a few characters in an
executable file, without corrupting it. Vim doesn't remove NUL
characters (they are represented as <NL> internally).
-b   command-line argument to start editing a binary file
'binary'   Option set by -b. Prevents adding an <EOL> for the
                last line in the file.

Multi-language support. multi-lang
Files in double-byte or multibyte encodings can be edited. There is
UTF-8 support to be able to edit various languages at the same time,
without switching fonts. UTF-8
Messages and menus are available in different languages.

Move cursor beyond lines.
When the 'virtualedit' option is set the cursor can move all over the
screen, also where there is no text. This is useful to edit tables
and figures easily.

Differences between Nvim and Vim

Nvim differs from Vim in many ways, although editor and Vimscript (not Vim9script) features are mostly identical. This document is a complete and centralized reference of the differences.

Configuration

nvim-config

User configuration and data files are found in standard base-directories (see also $NVIM_APPNAME). Note in particular:

- Use `$XDG_CONFIG_HOME/nvim/init.vim` instead of `.vimrc` for your config.
- Use `$XDG_CONFIG_HOME/nvim` instead of `.vim` to store configuration files.
- Use `$XDG_STATE_HOME/nvim/shada/main.shada` instead of `.viminfo` for persistent session information. shada

Defaults

defaults **nvim-defaults**

- Filetype detection is enabled by default. This can be disabled by adding ":filetype off" to init.vim.
- Syntax highlighting is enabled by default. This can be disabled by adding ":syntax off" to init.vim.
- Default color scheme has been updated. This can result in color schemes looking differently due to them relying on how highlight groups are defined by default. Add ":colorscheme vim" to init.vim or ":source $VIMRUNTIME/colors/vim.lua" to your color scheme file to restore the old default links and colors. See :highlight for a list of highlight groups colorschemes should set.
- 'autoindent' is enabled
- 'autoread' is enabled (works in all UIs, including terminal)
- 'background' defaults to "dark" (unless set automatically by the terminal/UI)
- 'backupdir' defaults to .,~/.local/state/nvim/backup// (xdg), auto-created
- 'belloff' defaults to "all"
- 'comments' includes "fb:•"
- 'commentstring' defaults to ""
- 'compatible' is always disabled
- 'complete' excludes "i"
- 'completeopt' defaults to "menu,popup"
- 'define' defaults to "". The C ftplugin sets it to "^\\s*#\\s*define"
- 'diffopt' includes "linematch:40"
- 'directory' defaults to ~/.local/state/nvim/swap// (xdg), auto-created
- 'display' defaults to "lastline"
- 'encoding' is UTF-8 (cf. 'fileencoding' for file-content encoding)
- 'fillchars' defaults (in effect) to "vert:│,fold:·,foldsep:│"
- 'formatoptions' defaults to "tcqj"
- 'grepprg' uses the -H and -I flags for regular grep, and defaults to using ripgrep if available
- 'hidden' is enabled
- 'history' defaults to 10000 (the maximum)
- 'hlsearch' is enabled
- 'include' defaults to "". The C ftplugin sets it to "^\\s*#\\s*include"
- 'incsearch' is enabled
- 'isfname' does not include ":" (on Windows). Drive letters are handled correctly without it. (Use gF for filepaths suffixed with ":line:col").
- 'joinspaces' is disabled
- 'jumpoptions' defaults to "clean"
- 'langnoremap' is enabled
- 'langremap' is disabled
- 'laststatus' defaults to 2 (statusline is always shown)
- 'listchars' defaults to "tab:> ,trail:-,nbsp:+"
- 'maxsearchcount' defaults to 999
- 'mouse' defaults to "nvi", see default-mouse for details
- 'mousemodel' defaults to "popup_setpos"
- 'nrformats' defaults to "bin,hex"
- 'path' defaults to ".,,". The C ftplugin adds "/usr/include" if it exists.
- 'sessionoptions' includes "unix,slash", excludes "options"
- 'shelltemp' is disabled
- 'shortmess' includes "CF", excludes "S"
- 'sidescroll' defaults to 1
- 'smarttab' is enabled
- 'spellfile' defaults to `stdpath("data")..."/site/spell/"`
- 'startofline' is disabled
- 'switchbuf' defaults to "uselast"
- 'tabpagemax' defaults to 50
- 'tags' defaults to "./tags;,tags"
- 'termguicolors' is enabled by default if Nvim can detect support from the host terminal
- 'ttimeout' is enabled
- 'ttimeoutlen' defaults to 50
- 'ttyfast' is set by default, but it can be unset during startup by setting the environment variable `NVIM_NOTTYFAST` to adjust the startup sequence for slow environments (such as ssh).
- 'undodir' defaults to ~/.local/state/nvim/undo// (xdg), auto-created
- 'viewoptions' includes "unix,slash", excludes "options"
- 'viminfo' includes "!"
- 'wildoptions' defaults to "pum,tagfile"
- dir plugin is enabled, so editing a directory shows its contents in a buffer with 'filetype' set to "directory".
- editorconfig plugin is enabled, .editorconfig settings are applied.
- man.lua plugin is enabled, so :Man is available by default.

- |matchit| plugin is enabled. To disable it in your config:

```
:let loaded_matchit = 1
```

- package-spellfile plugin is enabled, spellfiles are installed by default if missing.
- g:vimsyn_embed defaults to "l" to enable Lua highlighting

DEFAULT MOUSE

default-mouse disable-mouse By default the mouse is enabled. This means scroll-mouse-wheel will scroll the window instead of moving the cursor; <LeftMouse> click places the cursor; and <RightMouse> click opens the default popup-menu with standard actions.

Mouse is NOT enabled in Cmdline-mode or the more-prompt, so you can temporarily disable it just by typing ":". Or if you want to partially or fully disable the mouse or popup-menu, do any of the following:

- Disable mouse completely by unsetting the 'mouse' option:

```
set mouse=
```

- Change the 'mousemodel', so <RightMouse> extends selection instead of showing the popup-menu:

```
set mousemodel=extend
```

- Map <A-LeftMouse> so that it temporarily disables mouse until the cursor moves:

```
nnoremap <A-LeftMouse> <Cmd>
  \ set mouse=<Bar>
  \ echo 'mouse OFF until next cursor-move'<Bar>
  \ autocmd CursorMoved * ++once set mouse&<Bar>
  \ echo 'mouse ON'<CR>
```

To remove the default popup-menu without disabling mouse:

```
aunmenu PopUp
autocmd! nvim.popupmenu
```

To remove only the "How-to disable mouse" menu item (and its separator):

```
aunmenu PopUp.How-to\ disable\ mouse
aunmenu PopUp.-2-
```

DEFAULT MAPPINGS

default-mappings

Nvim creates the following default mappings at startup. You can disable any of these in your config by simply removing the mapping, e.g. ":unmap Y". If you never want any default mappings, call :mapclear early in your config.

- Y Y-default
- <C-U> i_CTRL-U-default
- <C-W> i_CTRL-W-default
- <C-L> CTRL-L-default
- & &-default
- - dir-mappings
- @ v_@-default
- # v_#-default
- * v_star-default
- gc gc-default v_gc-default o_gc-default
- gcc gcc-default
- gO
-]d]d-default
- [d [d-default
- [D [D-default
-]D]D-default
- <C-W>d CTRL-W_d-default
- [q]q [Q]Q [_CTRL-Q]_CTRL-Q
- [l]l [L]L [_CTRL-L]_CTRL-L
- [t]t [T]T [_CTRL-T]_CTRL-T
- [a]a [A]A
- [b]b [B]B
- [<Space>]<Space>
- LSP defaults lsp-defaults
  - K K-lsp-default
  - gr prefix gr-default
    - gra gri grn grr grt
  - <C-S> i_CTRL-S
- Treesitter defaults treesitter-defaults
  - v_an v_in
  - v_]n v_[n

DEFAULT AUTOCOMMANDS

default-autocmds

Default autocommands exist in the following groups. Use ":autocmd! {group}" to remove them and ":autocmd {group}" to see how they're defined.

nvim.terminal:

- BufReadCmd: Treats "term://" buffers as terminal buffers. terminal-start
- TermClose: A terminal buffer started with no arguments (which thus uses 'shell') and which exits with no error is closed automatically.
- TermClose: Displays the "[Process exited]" virtual text.
- TermRequest: The terminal emulator responds to OSC background and foreground requests, indicating (1) a black background and white foreground when Nvim option 'background' is "dark" or (2) a white background and black foreground when 'background' is "light". While this may not reflect the actual foreground/background color, it permits 'background' to be retained for a nested Nvim instance running in the terminal emulator.
- TermRequest: Nvim will create extmarks for shells which annotate their prompts with OSC 133 escape sequences, enabling users to quickly navigate between prompts using [[and]].
- TermOpen: Sets default options and mappings for terminal buffers:
  - 'nomodifiable'
  - 'undolevels' set to -1
  - 'textwidth' set to 0
  - 'nowrap'
  - 'nolist'
  - 'nonumber'
  - 'norelativenumber'
  - 'signcolumn' set to "no"
  - 'foldcolumn' set to "0"
  - 'winhighlight' uses hl-StatusLineTerm and hl-StatusLineTermNC in place of hl-StatusLine and hl-StatusLineNC
  - [[and]] to navigate between shell prompts

nvim.cmdwin:

- CmdwinEnter: Limits syntax sync to maxlines=1 in the cmdwin.

nvim.swapfile:

- SwapExists: Skips the swapfile prompt (sets v:swapchoice to "e") when the swapfile is owned by a running Nvim process. Shows W325 "Ignoring swapfile..." message.

nvim.exrc:

- VimEnter: Extend 'exrc' to also search for project-local configuration files in all parent directories.

nvim.progress:

- Progress: Display native progress bars in the TUI using the OSC 9;4 escape sequence.

New Features

nvim-features

MAJOR COMPONENTS

- API API
- Command atoms CmdAtom
- Job control job-control
- LSP framework lsp
- Lua scripting lua -l
- Multicursor multicursor
- Parsing engine treesitter
- Plugin manager vim.pack
- Providers
  - Clipboard provider-clipboard
  - Node.js plugins provider-nodejs
  - Python plugins provider-python
  - Ruby plugins provider-ruby
- Remote --remote-ui :connect :detach
- Remote plugins remote-plugin
- Shared data shada
- Terminal emulator terminal
- UI ui --listen --server
- Vimscript parser nvim_parse_expression()
- XDG base directories xdg

USER EXPERIENCE

Working intuitively and consistently is a major goal of Nvim.

feature-compile

- Nvim always includes ALL features, in contrast to Vim (which ships various combinations of 100+ optional features). feature-compile Think of it as a leaner version of Vim's "HUGE" build. This reduces surface area for bugs, and removes a common source of confusion and friction for users.
- Nvim avoids features that cannot be provided on all platforms; instead that is delegated to external plugins/extensions. E.g. the -X platform-specific option is "sometimes" available in Vim (with potential surprises: https://stackoverflow.com/q/14635295).
- Vim's internal test functions (test_autochdir(), test_settime(), etc.) are not exposed (nor implemented); instead Nvim has a robust API.
- Behaviors, options, documentation are removed if they cost users more time than they save.

Usability details have been improved where the benefit outweighs any backwards-compatibility cost. Some examples:

- Directories for 'directory' and 'undodir' are auto-created.
- Terminal features such as 'guicursor' are enabled where possible.
- Various "nvim" cli-arguments were redesigned.

Some features are built in that otherwise required external plugins:

- Highlighting the yanked region, see vim.hl.

ARCHITECTURE

The Nvim UI is "decoupled" from the core editor: all UIs, including the builtin TUI are just plugins that connect to a Nvim server (via --server or --embed). Multiple UI clients can :connect to the same editor server.

External plugins run in separate processes. remote-plugin This improves stability and allows those plugins to work without blocking the editor. Even "legacy" Python and Ruby plugins which use the old Vim interfaces (if_pyth, if_ruby) run out-of-process, so they cannot crash Nvim.

Platform and I/O facilities are built upon libuv. Nvim benefits from libuv features and bug fixes, and other projects benefit from improvements to libuv by Nvim developers. This enables features such as vim.wait() and vim.uv.

FEATURES

Command-line:
- The expression prompt (@=, c_CTRL-R_=, i_CTRL-R_=) is highlighted using a built-in Vimscript expression parser. expr-highlight
- E5408 E5409 input(), inputdialog() support custom highlighting. input()-highlight
- (Experimental) g:Nvim_color_cmdline Command-line (:) is colored by callback defined in `g:Nvim_color_cmdline` (this callback is for testing only, and will be removed in the future).
- cmdwin (q:, q/, q?, c_CTRL-F) is a "normal" buffer+window, instead of a nested modal loop. API and Lua callbacks can run while it is open. The cmdwin-char is shown via 'statuscolumn'.

Commands:
- :bcd
- :checkhealth
- :connect
- :detach
- :diffget, :diffput (do, dp) can be . repeated.
- :drop is always available
- :EditQuery
- :Inspect
- :InspectTree
- :log
- :lsp
- :Man is available by default, with many improvements such as completion
- :match can be invoked before highlight group is defined
- :packdel
- :packupdate
- :restart
- :source works with Lua
- :trust
- User commands can support :command-preview to show results as you type
- :write with "++p" flag creates parent directories.
- :update command writes new file buffers even when unmodified.
- :uptime

Editor:
- ins-repeat replays the full Insert session, including cursor-moves: "iab<Left>c<Esc>." produces "acb", not "c".
- visual-repeat is "semantic" instead of "fixed-size". If you prefer Vim's "fixed-size" Visual-repeat, see visual-fixed-size.
- Interactive Ex-mode is implemented as persistent, insert-mode cmdwin.
- prompt-buffer supports multiline input/paste, undo/redo, and o/O normal commands.
- i_CTRL-R inserts named/clipboard registers (A-Z,a-z,0-9+) literally, like pasting instead of like user input. Improves performance, avoids broken formatting. To get the old behavior you can use `<C-R>=@x`.

Events (autocommands):
- Fixed inconsistent behavior in execution of nested autocommands #23368
- InsertCharPre is not triggered for stuffed text (., redo, i_CTRL-R, i_CTRL-A), which also skips mappings and abbreviations.
- OptionSet is triggered for 'modified' when writing a file, undoing changes, or using :set modified.
- Progress
- RecordingEnter
- RecordingLeave
- SearchWrapped
- Signal
- TabMoved
- TabNewEntered
- TermClose
- TermOpen
- TermRequest
- TermResponse is fired for DCS, OSC, and APC sequences received from the terminal, in addition to the Primary Device Attributes response. v:termresponse
- UIEnter
- UILeave

Functions:
- dictwatcheradd() notifies a callback whenever a Dict is modified

- dictwatcherdel()
- menu_get()
- msgpackdump(), msgpackparse() provide msgpack de/serialization
- stdpath()
- system(), systemlist() can run {cmd} directly (without 'shell')
- matchadd() can be called before highlight group is defined
- tempname() tries to recover if the Nvim tempdir disappears.
- writefile() with "p" flag creates parent directories.
- prompt_getinput()
- prompt_appendbuf()

Highlight groups:
- highlight-blend controls blend level for a highlight group
- expr-highlight highlight groups (prefixed with "Nvim")
- hl-NormalFloat highlights floating window
- hl-FloatBorder highlights border of a floating window
- hl-FloatTitle highlights title of a floating window
- hl-FloatFooter highlights footer of a floating window
- hl-NormalNC highlights non-current windows
- hl-MsgArea highlights messages/cmdline area
- hl-MsgSeparator highlights separator for scrolled messages
- hl-OkMsg
- hl-StderrMsg
- hl-StdoutMsg
- hl-Substitute
- hl-TermCursor
- hl-WinSeparator highlights window separators
- hl-Whitespace highlights 'listchars' whitespace
- hl-WinBar highlights 'winbar'
- hl-WinBarNC highlights non-current window 'winbar'

Input/Mappings:
- ALT (META) chords always work (even in the TUI). Map <M- with any key: <M-1>, <M-BS>, <M-Del>, <M-Ins>, <M-/>, <M-\>, <M-Space>, <M-Enter>, etc.
  - Case-sensitive: <M-a> and <M-A> are two different keycodes.
- ALT may behave like <Esc> if not mapped. i_ALT v_ALT c_ALT

Normal commands:
- gO shows a filetype-defined "outline" of the current buffer.
- gQ restores multicursors.
- Q adds a multicursor instead of switching to Ex mode.
- [count]q: enters Ex-mode.
- ZR performs :restart
- v_al selects the whole buffer; v_il selects the current line without leading or trailing white space.

Options:

- :set {option}< removes local value for all global-local options.
- :setlocal {option}< copies global value to local value for all options.
- 'ambiwidth' cannot be set to empty.
- 'autoread' works in the terminal (if it supports "focus" events)
- 'background' cannot be set to empty.
- 'completeopt' flag "preselect" selects first completion item that has the "preselect" attribute set.
- 'cpoptions' flags: cpo-_
- 'eadirection' cannot be set to empty.
- 'exrc' searches for ".nvim.lua", ".nvimrc", or ".exrc" files. The user is prompted whether to trust the file.
- 'fileformat' cannot be set to empty.
- 'fillchars' flags: "msgsep", "horiz", "horizup", "horizdown", "vertleft", "vertright", "verthoriz"
- 'foldcolumn' supports up to 9 dynamic/fixed columns
- 'guicursor' works in the terminal (TUI)
- 'inccommand' shows interactive results for :substitute-like commands and :command-preview commands
- 'isident' does not affect environment variable name parsing. expr-env
- 'jumpoptions'
  - "view" tries to restore mark-view when moving through the jumplist, changelist, alternate-file, and mark-motions.
  - "clean" removes unloaded buffers from the jumplist.
- 'langmap' supports multibyte chars.
- 'laststatus' global statusline support
- 'mousemodel' cannot be set to empty.
- 'mousescroll' amount to scroll by when scrolling with a mouse
- 'pumblend' pseudo-transparent popupmenu
- 'scrollback'
- 'shortmess'
  - "F" flag does not affect output from autocommands.
  - "q" flag fully hides macro recording message.
- 'showcmdloc' cannot be set to empty.
- 'signcolumn' can show multiple signs (dynamic or fixed columns)
- 'statuscolumn' full control of columns using 'statusline' format
- 'statusline' default is exposed as a statusline expression.
- 'statusline' and related options apply %= and %< to item groups.
- 'splitkeep' cannot be set to empty.

- 'tabline' middle-click on tabpage label closes tabpage, and %@Func@foo%X can call any function on mouse-click
- 'termpastefilter'
- 'ttimeout', 'ttimeoutlen' behavior was simplified
- 'winblend' pseudo-transparency in floating windows api-floatwin
- 'winhighlight' window-local highlights
- 'busy' busy status for buffers
- 'messagesopt' added progress:c to control if progress messages are shown in cmdline

Performance:
- Signs are implemented using Nvim's internal "marktree" (btree) structure.
- Folds are not updated during insert-mode.

Providers:
- If a Python interpreter is available on your $PATH, :python and :python3 are always available. See provider-python.

Shell:
- Shell output (:!, :make, ...) is always routed through the UI, so it cannot "mess up" the screen. (You can still use "chansend(v:stderr,...)" if you want to mess up the screen :)
- Nvim throttles (skips) messages from shell commands (:!, :grep, :make) if there is too much output. No data is lost, this only affects display and improves performance. :terminal output is never throttled.
- :! does not support "interactive" commands. Use :terminal instead. (GUI Vim has a similar limitation, see ":help gui-pty" in Vim.)
- :!start is not special-cased on Windows.
- system() does not support writing/reading "backgrounded" commands. E5677

Signs:
- 'signcolumn' can show multiple signs.
- Signs are removed if the associated line is deleted.
- Signs placed twice with the same identifier in the same group are moved.

Startup:
- -e and -es invoke the same "improved Ex mode" as -E and -Es.
- -E and -Es read stdin as text (into buffer 1).
- -es and -Es act as a generic "script mode":
  - Exits automatically, don't need "-c qa!".
  - Skips swap-file dialog.
  - Disables swapfile, shada.
  - Various "POSIX Ex" quirks were dropped.
- -l Executes Lua scripts non-interactively.
- -s reads Normal commands from stdin if the script name is "-".
- Reading text (instead of commands) from stdin --:
  - works by default: "-" file is optional
  - works in more cases: -Es, file args

TUI: :set-termcap
- Start Nvim with 'verbose' level 3 to show terminal capabilities:

```
nvim -V3
```

'term' E529 E530 E531
- 'term' reflects the terminal type derived from $TERM and other environment checks. Use `:echo &term` to get its value. For debugging only; not reliable during startup.
  - **Note:** If you want to detect when Nvim is running in a terminal, use `has('gui_running')` has() or see nvim_list_uis() for an example of how to inspect the UI channel.
- Nvim might optionally be compiled with unibilium, in which case the terminfo database will be used. Otherwise, or if the terminal was not found in the database, a table of builtin terminal definitions will be used.
- Nvim will use 256-colour capability on Linux virtual terminals. Vim uses only 8 colours plus bright foreground on Linux VTs.
- Vim combines what is in its builtin-terms with what it reads from terminfo, and has a 'ttybuiltin' setting to control how that combination works. Nvim uses one or the other, it does not attempt to merge the two.

UI/Display:
- messages: When showing messages longer than 'cmdheight', only scroll the message lines, not the entire screen. The separator line is decorated by hl-MsgSeparator and the "msgsep" flag of 'fillchars'. msgsep

Variables:
- v:progpath is always absolute ("full")
- v:windowid is always available (for use by external UIs)
- OptionSet autocommand args v:option_new, v:option_old,
- v:option_oldlocal, v:option_oldglobal have the type of the option instead of always being strings. v:option_old is now the old global value for all global-local options, instead of just string global-local options.

Vimscript:
- :redir nested in execute() works.

Upstreamed features nvim-upstreamed

These Nvim features were later integrated into Vim.

- 'diffopt' "linematch" feature
- 'fillchars' flags: "eob"
- 'jumpoptions' "stack" behavior
- 'wildoptions' flags: "pum" enables popupmenu for wildmode completion
- <Cmd>
- WinClosed

- WinScrolled
- :sign-define "numhl" argument
- :source works with anonymous (no file) scripts
- 'statusline' supports unlimited alignment sections
- vim-tutor-mode
- Lua: `require()` also searches for modules in 'runtimepath'.
- Lua: vim.g, vim.w, vim.b, vim.t, vim.v, vim.fn, vim.call()
- Visual selection highlights the character at cursor. visual-use

Other changes

nvim-changed

This section documents various low-level behavior changes.

mkdir() behaviour changed:
- 1. Assuming /tmp/foo does not exist and /tmp can be written to mkdir('/tmp/foo/bar', 'p', 0700) will create both /tmp/foo and /tmp/foo/bar with 0700 permissions. Vim mkdir will create /tmp/foo with 0755.
- 2. If you try to create an existing directory with `'p'` (e.g. mkdir('/', 'p')) mkdir() will silently exit. In Vim this was an error.
- 3. mkdir() error messages now include strerror() text when mkdir fails.

string() and :echo behaviour changed:
- 1. No maximum recursion depth limit is applied to nested container structures.
- 2. string() fails immediately on nested containers, not when recursion limit was exceeded.
- 3. When :echo encounters duplicate containers like

```
let l = []
echo [l, l]
```

  it does not use "[...]" (was: "[[], [...]]", now: "[[], []]"). "..." is only used for recursive containers.
- 4. :echo printing nested containers adds "@level" after "..." designating the level at which recursive container was printed: :echo-self-refer. Same thing applies to string() (though it uses construct like "{E724@level}"), but this is not reliable because string() continues to error out.
- 5. Stringifyed infinite and NaN values now use str2float() and can be evaled back.
- 6. (internal) Trying to print or stringify VAR_UNKNOWN in Vim results in nothing, E908, in Nvim it is internal error.

json_decode() behaviour changed:
- 1. It may output msgpack-special-dict.
- 2. msgpack-special-dict is emitted also in case of duplicate keys, while in Vim it errors out.
- 3. It accepts only valid JSON. Trailing commas are not accepted.

json_encode() behaviour slightly changed: now msgpack-special-dict values are accepted, but v:none is not.

Viminfo text files were replaced with binary (messagepack) shada files. Additional differences:

- shada-c has no effect.
- shada-s now limits size of every item and not just registers.
- 'viminfo' option got renamed to 'shada'. Old option is kept as an alias for compatibility reasons.
- :wviminfo was renamed to :wshada, :rviminfo to :rshada. Old commands are still kept.
- ShaDa file format was designed with forward and backward compatibility in mind. shada-compatibility
- Some errors make ShaDa code keep temporary file in-place for user to decide what to do with it. Vim deletes temporary file in these cases. shada-error-handling
- ShaDa file keeps search direction (v:searchforward), viminfo does not.

printf() returns something meaningful when used with `%p` argument: in Vim it used to return useless address of the string (strings are copied to the newly allocated memory all over the place) and fail on types which cannot be coerced to strings. See id() for more details, currently it uses `printf("%p", {expr})` internally.

c_CTRL-R pasting a non-special register into cmdline omits the last <CR>.

CursorMoved triggers when moving between windows.

Lua interface (lua.txt):
- `:lua print("a\0b")` will print a^@b, like with `:echomsg "a\nb"` . In Vim that prints a and b on separate lines, exactly like `:lua print("a\nb")` .
- `:lua error('TEST')` emits the error:

```
E5108: Lua: [string "<Vimscript compiled string>"]:1: TEST
```

  whereas Vim emits only "TEST".
- Lua has direct access to Nvim API via `vim.api`.

Commands:
- :doautocmd does not warn about "No matching autocommands".
- :wincmd accepts a count.
- `:write!` does not show a prompt if the file was updated externally.
- := does not accept ex-flags. With an arg it is equivalent to :lua=

Command-line:
- The meanings of arrow keys do not change depending on 'wildoptions'.

Functions:
- input() and inputdialog() support for each other's features (return on cancel and completion respectively) via dictionary argument (replaces all other arguments if used), and "cancelreturn" can have any type if passed in a dictionary.
- input() and inputdialog() support user-defined cmdline highlighting.

Highlight groups:
- hl-ColorColumn, hl-CursorColumn are lower priority than most other groups

- hl-CursorLine is low-priority unless foreground color is set
- hl-VertSplit superseded by hl-WinSeparator
- Highlight groups names are allowed to contain `@` characters.
  - It is an error to define a highlight group with a name that doesn't match the regexp `[a-zA-Z0-9_.@-]*` (see group-name).
- hl-StatusLineTerm hl-StatusLineTermNC are implemented as 'winhighlight' window-local highlights which are set by the default TermOpen handler.
- The ins-completion-menu has cascading highlight styles. hl-PmenuSel and hl-PmenuMatch both inherit from hl-Pmenu, and hl-PmenuMatchSel inherits highlights from both hl-PmenuSel and hl-PmenuMatch.

Macro (recording) behavior:
- Replay of a macro recorded during :lmap produces the same actions as when it was recorded. In Vim if a macro is recorded while using :lmap'ped keys then the behaviour during record and replay differs.
- 'keymap' is implemented via :lmap instead of :lnoremap so that you can use macros and 'keymap' at the same time. This also means you can use :imap on the results of keys from 'keymap'.

Mappings:
- Creating a mapping for a simplifiable key (e.g. <C-I>) doesn't replace an existing mapping for its simplified form (e.g. <Tab>).
- "#" followed by a digit doesn't stand for a function key at the start of the lhs of a mapping.

Marks:
- undo does not restore a mark that you moved (m, :mark) after the change being undone; the mark shifts with the text it was moved to, the same as a mark the change never touched. Vim restores its old (wrong) position. Marks moved by the change itself are still restored.
- An operator on a target that creates a Visual selection ("cgn") does not change '< '>. gv

Motion:
- The jumplist avoids useless/phantom jumps.

Syntax highlighting:
- syncolor.vim has been removed. Nvim now sets up default highlighting groups automatically for both light and dark backgrounds, regardless of whether or not syntax highlighting is enabled. This means that :syntax-on and :syntax-enable are now identical. Users who previously used an after/syntax/syncolor.vim file should transition that file into a colorscheme. :colorscheme

Vimscript compatibility:
- `count` does not alias to v:count
- `errmsg` does not alias to v:errmsg
- `shell_error` does not alias to v:shell_error
- `this_session` does not alias to v:this_session

Working directory (Vim implemented some of these after Nvim):
- DirChanged and DirChangedPre can be triggered when switching to another window or tab.
- getcwd() and haslocaldir() may throw errors if the tabpage, window, or buffer cannot be found.
- haslocaldir() checks for tab-local directory if and only if `-1` is passed as window number, and its only possible returns values are 0 and 1.
- `getcwd(-1)` is equivalent to `getcwd(-1, 0)` instead of returning the global working directory. Use `getcwd(-1, -1)` to get the global working directory.

Options:
- 'titlestring' uses printf-style '%' items (see: 'statusline') to implement the default behaviour. The implementation is equivalent to setting 'titlestring' to `%t%(\ %M%)%(\ \(%{expand(\"%:~:h\")}\)%)%a\ -\ Nvim`.

packages:
- "start" packages are not explicitly reported in 'runtimepath' (though they are still searched for runtime files). See runtime-search-path

Missing features

nvim-missing

These legacy Vim features are not yet implemented:

- :gui
- 'browsedir' 'bsdir'
- 'completepopup'
- 'guioptions' 'go'
- 'guitablabel' 'gtl'
- 'guitabtooltip' 'gtt'
- 'mouseshape' 'mouses'

Removed legacy features

nvim-removed

These Vim features were intentionally removed from Nvim.

Aliases:
- ex (alias for "nvim -e")
- exim (alias for "nvim -E")
- gex (GUI)
- gview (GUI)
- gvim (GUI)
- gvimdiff (GUI)
- rgview (GUI)
- rgvim (GUI)
- rview
- rvim
- view (alias for "nvim -R")
- vimdiff (alias for "nvim -d" diff-mode)

Commands:

- :behave
- :fixdel
- :gvim
- hardcopy `:hardcopy` was removed. Instead, use `:TOhtml` and print the resulting HTML using a web browser or other HTML viewer.
- :helpfind
- :mode (no longer accepts an argument)
- :open
- :Print
- :promptfind
- :promptrepl
- :scriptversion (always version 1)
- :shell
- :smile
- :tearoff
- :cstag
- :cscope
- :lcscope
- :scscope
- :Vimuntar
- `:TOhtml` was replaced by a Lua version (with various differences)

Compile-time features:
- Emacs tags support
- X11 integration (see x11-selection)

Editor:
- c_CTRL-F cannot enter cmdwin from the expr-register prompt.
- cscope support was removed in favour of plugin-based solutions such as: https://github.com/dhananjaylatkar/cscope_maps.nvim
- popup-window : Use floating-windows instead.
- spellfile.vim : Replaced by package-spellfile.
- textprop : Use extmarks instead.

Eval:
- Vim9script (the Vim 9+ flavor of Vimscript) is not supported.
- cscope_connection()
- err_teapot()
- js_encode()
- js_decode()
- listener_add() , etc: use api-buffer-updates-lua instead.
- redraw_listener_add() , etc: use nvim_set_decoration_provider() instead.
- v:none (used by Vim to represent JavaScript "undefined"); use v:null instead.
- v:sizeofint
- v:sizeoflong
- v:sizeofpointer

Events:
- SafeStateAgain
- SigUSR1 Use Signal to detect `SIGUSR1` signal instead.

Options:
- 'aleph' 'al'
- antialias
- 'backspace' no longer supports number values. Instead:
  - for `backspace=0` set `backspace=` (empty)
  - for `backspace=1` set `backspace=indent,eol`
  - for `backspace=2` set `backspace=indent,eol,start` (default behavior in Nvim)
  - for `backspace=3` set `backspace=indent,eol,nostop`
- 'balloondelay' 'bdlay'
- 'balooneval' 'beval' 'noballooneval' 'nobeval'
- 'balloonexpr' 'bexpr'
- bioskey (MS-DOS)
- conskey (MS-DOS)
- 'cp' 'nocompatible' 'nocp' 'compatible' (Nvim is always "nocompatible".)
- 'cpoptions' (gjpkHw<*- and all POSIX flags were removed)
- 'cryptmethod' 'cm' 'key' (Vim encryption implementation)
- cscopepathcomp
- cscopeprg
- cscopequickfix
- cscoperelative
- cscopetag
- cscopetagorder
- cscopeverbose
- 'ed' 'edcompatible' 'noed' 'noedcompatible'
- 'encoding' ("utf-8" is always used)
- esckeys
- 'guioptions' "t" flag was removed
- 'guifontset' 'gfs' (Use 'guifont' instead.)
- 'guipty' (Nvim uses pipes and PTYs consistently on all platforms.)
- 'highlight' (Names of builtin highlight-groups cannot be changed.)
- 'hkmap' 'hk' use `set keymap=hebrew` instead.

- 'hkmapp' 'hkp' use `set keymap=hebrewp` instead.
- keyprotocol
- 'pastetoggle' 'pt' Just Paste It.™ paste is handled automatically when you paste text using your terminal's or GUI's paste feature (`CTRL-SHIFT-v`, CMD-v (macOS), middle-click, ...).
- 'imactivatefunc' 'imaf'
- 'imactivatekey' 'imak'
- 'imcmdline' 'imc' 'noimcmdline' 'noimc'
- 'imdisable' 'imd' 'noimdisable' 'noimd'
- 'imstatusfunc' 'imsf'
- 'insertmode' 'im' Use the following script to emulate 'insertmode':

```
autocmd BufWinEnter * startinsert
inoremap <Esc> <C-X><C-Z><C-]>
inoremap <C-C> <C-X><C-Z>
inoremap <C-L> <C-X><C-Z><C-]><Esc>
inoremap <C-Z> <C-X><C-Z><Cmd>suspend<CR>
noremap <C-C> <Esc>
snoremap <C-C> <Esc>
noremap <C-\><C-G> <C-\><C-N><Cmd>startinsert<CR>
cnoremap <C-\><C-G> <C-\><C-N><Cmd>startinsert<CR>
inoremap <C-\><C-G> <C-X><C-Z>
autocmd CmdwinEnter * noremap <buffer> <C-C> <C-C>
autocmd CmdwinEnter * inoremap <buffer> <C-C> <C-C>

lua << EOF
  vim.on_key(function(c)
    if c == '\27' then
      local mode = vim.api.nvim_get_mode().mode
      if mode:find('^[nvV\22sS\19]') and vim.fn.getcmdtype() == '' then
        vim.schedule(function()
          vim.cmd('startinsert')
        end)
      end
    end
  end)
EOF
```

- 'macatsui'
- 'maxcombine' 'mco' : Nvim counts maximum character sizes in bytes, not codepoints. This is guaranteed to be big enough to always fit all chars properly displayed in vim with 'maxcombine' set to 6.
  - You can still edit text with larger characters than fits in the screen buffer, you just can't see them. Use g8 or ga. See mbyte-combining.
  - **NOTE:** the rexexp engine still has a hard-coded limit of considering 6 composing chars only.
- 'maxmem' Nvim delegates memory-management to the OS.
- 'maxmemtot' Nvim delegates memory-management to the OS.
- 'opendevice' 'odev' 'noopendevice' 'noodev'
- printoptions
- 'printdevice'
- 'printencoding'
- 'printexpr'
- 'printfont'
- 'printheader'
- 'printmbcharset'
- 'prompt' 'noprompt'
- 'remap' 'noremap'
- 'restorescreen' 'rs' 'norestorescreen' 'nors'
- 'secure' : Everything is allowed in 'exrc' files, because they must be explicitly marked as "trusted".
- 'shelltype'
- 'shortmess' flags: shm-f shm-n shm-x shm-i (behave like always on)
- 'shortname' 'sn' 'noshortname' 'nosn'
- 'swapsync' 'sws'
- 'termencoding' 'tenc' (Vim 7.4.852 also removed this for Windows)
- 'terse' 'noterse' (Add "s" to 'shortmess' instead)
- textauto
- textmode
- 'toolbar' 'tb'
- 'toolbariconsize' 'tbis'
- 'ttybuiltin' 'tbi' 'nottybuiltin' 'notbi'
- 'ttymouse' 'ttym'
- 'ttyscroll' 'tsl'
- 'ttytype' 'tty'
- weirdinvert

Plugins:
- logiPat
- rrhelper
- vimball
- shellmenu
- macmap.vim
- tools/check_colors.vim
- macros/{justify,matchit,swapmous}.vim: use `packadd! justify` etc. directly
- HelpToc: Use gO instead.

Providers:

- if_lua : Nvim Lua API is not compatible with Vim's "if_lua".
- if_mzscheme
- if_pyth: python-bindeval python-Function are not supported.
- if_tcl

Startup:

- `--literal`: File args are always literal; to expand wildcards on Windows, use :n e.g. `nvim +"n *"`
- --serverlist Use instead: `nvim -es -V1 +'echo serverlist()'`
- --servername Use instead: --server or :connect.
- Easy mode: eview, evim, nvim -y
- Restricted mode: rview, rvim, nvim -Z
- Vi mode: nvim -v

Test functions:

- test_alloc_fail()
- test_autochdir()
- test_feedinput()
- test_garbagecollect_soon()
- test_getvalue()
- test_ignore_error()
- test_null_blob()
- test_null_channel()
- test_null_dict()
- test_null_function()
- test_null_job()
- test_null_list()
- test_null_partial()
- test_null_string()
- test_option_not_set()
- test_override()
- test_refcount()
- test_scrollbar()
- test_setmouse()
- test_settime()
- test_srand_seed()
- test_unknown()
- test_void()

TUI: t_xx termcap-options t_AB t_Sb t_vb t_SI

- Nvim does not have special t_XX options nor <t_XX> keycodes to configure terminal capabilities. Instead Nvim treats the terminal as any other UI, e.g. 'guicursor' sets the terminal cursor style if possible. You can use $NVIM_TERMDEFS if you still need to override the builtin definitions.

termcap

- Nvim never uses the termcap database, only terminfo and builtin-terms.

xterm-8bit xterm-8-bit

- Xterm can be run in a mode where it uses true 8-bit CSI. Supporting this requires autodetection of whether the terminal is in UTF-8 mode or non-UTF-8 mode, as the 8-bit CSI character has to be written differently in each case. Vim issues a "request version" sequence to the terminal at startup and looks at how the terminal is sending CSI. Nvim does not issue such a sequence and always uses 7-bit control sequences.

Expression evaluation expr E15 eval eval.txt

Using expressions is introduced in chapter 41 of the user manual usr_41.txt.

1. Variables variables

1.1 Variable types

E712 E896 E897 E899 E908 E909
E928 E964 E966 E1023 E1098 E1174 E1175
E1203 E1206 E1210 E1211 E1212 E1219 E1220
E1222 E1225 E1226 E1238 E1250 E1252 E1256
E1265 E1297 E5010 E5050 E5060 E5070 E5071
E5299 E5300 E5401 E5420 E6000

There are seven types of variables:

Number Integer

Number	A 32 or 64 bit signed number. expr-number The number of bits is available in v:numbersize. Examples: -123 0x10 0177 0o177 0b1011
Float	A floating point number. floating-point-format Float Examples: 123.456 1.15e-6 -1.1e3
String	A NUL terminated string of 8-bit unsigned characters (bytes). expr-string Examples: "ab\txx\"--" 'x-z''a,c'
Funcref	A reference to a function Funcref. Example: function("strlen") It can be bound to a dictionary and arguments, it then works like a Partial. Example: function("Callback", [arg], myDict)
List	An ordered sequence of items, see List for details. Example: [1, 2, ['a', 'b']]
Dictionary	An associative, unordered array: Each entry has a key and a value. Dictionary Examples:

```
{"blue": "#0000ff", "red": "#ff0000"}
#{blue: "#0000ff", red: "#ff0000"}
```

Blob	Binary Large Object. Stores any sequence of bytes. See Blob for details. Example: 0zFF00ED015DAF 0z is an empty Blob.

The Number and String types are converted automatically, depending on how they are used.

Conversion from a Number to a String is by making the ASCII representation of the Number. Examples:

Number 123 --> String "123"
Number 0 --> String "0"
Number -1 --> String "-1"

octal

Conversion from a String to a Number is done by converting the first digits to a number. Hexadecimal "0xf9", Octal "017" or "0o17", and Binary "0b10" numbers are recognized. If the String doesn't start with digits, the result is zero. Examples:

String "456" --> Number 456
String "6bar" --> Number 6
String "foo" --> Number 0
String "0xf1" --> Number 241
String "0100" --> Number 64
String "0o100" --> Number 64
String "0b101" --> Number 5
String "-8" --> Number -8
String "+8" --> Number 0

To force conversion from String to Number, add zero to it:

```
:echo "0100" + 0
```

64

To avoid a leading zero to cause octal conversion, or for using a different base, use str2nr().

TRUE FALSE Boolean

For boolean operators Numbers are used. Zero is FALSE, non-zero is TRUE.
You can also use v:false and v:true.
When TRUE is returned from a function it is the Number one, FALSE is the number zero.

Note that in the command:

```
:if "foo"
:" NOT executed
```

"foo" is converted to 0, which means FALSE. If the string starts with a non-zero number it means TRUE:

```
:if "8foo"
:" executed
```

To test for a non-empty string, use empty():

```
:if !empty("foo")
```

falsy truthy

An expression can be used as a condition, ignoring the type and only using whether the value is "sort of true" or "sort of false". Falsy is:

```
        the number zero
        empty string, blob, list or dictionary
```

Other values are truthy. Examples:

```
        0       falsy
        1       truthy
        -1      truthy
        0.0     falsy
        0.1     truthy
        ''      falsy
        'x'     truthy
        []      falsy
        [0]     truthy
        {}      falsy
        #{x: 1} truthy
        0z      falsy
        0z00    truthy
```

non-zero-arg

Function arguments often behave slightly different from TRUE: If the argument is present and it evaluates to a non-zero Number, v:true or a non-empty String, then the value is considered to be TRUE.
Note that " " and "0" are also non-empty strings, thus considered to be TRUE.
A List, Dictionary or Float is not a Number or String, thus evaluate to FALSE.

E745 E728 E703 E729 E730 E731
E974 E975 E976

List, Dictionary, Funcref, and Blob types are not automatically converted.

E805 E806 E808

When mixing Number and Float the Number is converted to Float. Otherwise there is no automatic conversion of Float. You can use str2float() for String to Float, printf() for Float to String and float2nr() for Float to Number.

E362 E891 E892 E893 E894 E907

When expecting a Float a Number can also be used, but nothing else.

no-type-checking

You will not get an error if you try to change the type of a variable.

1.2 Function references

Funcref E695 E718 E1192

A Funcref variable is obtained with the function() function, the funcref() function or created with the lambda expression expr-lambda. It can be used in an expression in the place of a function name, before the parenthesis around the arguments, to invoke the function it refers to. Example:

```
:let Fn = function("MyFunc")
:echo Fn()
```

E704 E705 E707

A Funcref variable must start with a capital, "s:", "w:", "t:" or "b:". You can use "g:" but the following name must still start with a capital. You cannot have both a Funcref variable and a function with the same name.

A special case is defining a function and directly assigning its Funcref to a Dictionary entry. Example:

```
:function dict.init() dict
:   let self.val = 0
:endfunction
```

The key of the Dictionary can start with a lower case letter. The actual function name is not used here. Also see numbered-function.

A Funcref can also be used with the :call command:

```
:call Fn()
:call dict.init()
```

The name of the referenced function can be obtained with string().

```
:let func = string(Fn)
```

You can use call() to invoke a Funcref and use a list variable for the arguments:

```
:let r = call(Fn, mylist)
```

Partial

A Funcref optionally binds a Dictionary and/or arguments. This is also called
a Partial. This is created by passing the Dictionary and/or arguments to
function() or funcref(). When calling the function the Dictionary and/or
arguments will be passed to the function. Example:

```
let Cb = function('Callback', ['foo'], myDict)
call Cb('bar')
```

This will invoke the function as if using:

```
call myDict.Callback('foo', 'bar')
```

Note that binding a function to a Dictionary also happens when the function is
a member of the Dictionary:

```
let myDict.myFunction = MyFunction
call myDict.myFunction()
```

Here MyFunction() will get myDict passed as "self". This happens when the
"myFunction" member is accessed. When assigning "myFunction" to otherDict
and calling it, it will be bound to otherDict:

```
let otherDict.myFunction = myDict.myFunction
call otherDict.myFunction()
```

Now "self" will be "otherDict". But when the dictionary was bound explicitly
this won't happen:

```
let myDict.myFunction = function(MyFunction, myDict)
let otherDict.myFunction = myDict.myFunction
call otherDict.myFunction()
```

Here "self" will be "myDict", because it was bound explicitly.

1.3 Lists

list List Lists E686

A List is an ordered sequence of items. An item can be of any type. Items
can be accessed by their index number. Items can be added and removed at any
position in the sequence.

List creation

E696 E697

A List is created with a comma-separated list of items in square brackets.
Examples:

```
:let mylist = [1, two, 3, "four"]
:let emptylist = []
```

An item can be any expression. Using a List for an item creates a
List of Lists:

```
:let nestlist = [[11, 12], [21, 22], [31, 32]]
```

An extra comma after the last item is ignored.

List index

list-index E684

An item in the List can be accessed by putting the index in square brackets
after the List. Indexes are zero-based, thus the first item has index zero.

```
:let item = mylist[0]           " get the first item: 1
:let item = mylist[2]           " get the third item: 3
```

When the resulting item is a list this can be repeated:

```
:let item = nestlist[0][1]      " get the first list, second item: 12
```

A negative index is counted from the end. Index -1 refers to the last item in
the List, -2 to the last but one item, etc.

```
:let last = mylist[-1]          " get the last item: "four"
```

To avoid an error for an invalid index use the get() function. When an item
is not available it returns zero or the default value you specify:

```
:echo get(mylist, idx)
:echo get(mylist, idx, "NONE")
```

List concatenation

list-concatenation

Two lists can be concatenated with the "+" operator:

```
:let longlist = mylist + [5, 6]
:let longlist = [5, 6] + mylist
```

To prepend or append an item, turn it into a list by putting [] around it.

A list can be concatenated with another one in-place using :let+= or extend():

```
:let mylist += [7, 8]
:call extend(mylist, [7, 8])
```

See list-modification below for more about changing a list in-place.

Sublist

sublist

A part of the List can be obtained by specifying the first and last index, separated by a colon in square brackets:

```
:let shortlist = mylist[2:-1]   " get List [3, "four"]
```

Omitting the first index is similar to zero. Omitting the last index is similar to -1.

```
:let endlist = mylist[2:]       " from item 2 to the end: [3, "four"]
:let shortlist = mylist[2:2]    " List with one item: [3]
:let otherlist = mylist[:]      " make a copy of the List
```

Notice that the last index is inclusive. If you prefer using an exclusive index use the slice() method.

If the first index is beyond the last item of the List or the second item is before the first item, the result is an empty list. There is no error message.

If the second index is equal to or greater than the length of the list the length minus one is used:

```
:let mylist = [0, 1, 2, 3]
:echo mylist[2:8]               " result: [2, 3]
```

NOTE: mylist[s:e] means using the variable "s:e" as index. Watch out for using a single letter variable before the ":". Insert a space when needed: mylist[s : e].

List identity

list-identity

When variable "aa" is a list and you assign it to another variable "bb", both variables refer to the same list. Thus changing the list "aa" will also change "bb":

```
:let aa = [1, 2, 3]
:let bb = aa
:call add(aa, 4)
:echo bb
```

```
        [1, 2, 3, 4]
```

Making a copy of a list is done with the copy() function. Using [:] also works, as explained above. This creates a shallow copy of the list: Changing a list item in the list will also change the item in the copied list:

```
:let aa = [[1, 'a'], 2, 3]
:let bb = copy(aa)
:call add(aa, 4)
:let aa[0][1] = 'aaa'
:echo aa
```

```
        [[1, aaa], 2, 3, 4]
```

```
:echo bb
```

```
        [[1, aaa], 2, 3]
```

To make a completely independent list use deepcopy(). This also makes a copy of the values in the list, recursively. Up to a hundred levels deep.

The operator "is" can be used to check if two variables refer to the same List. "isnot" does the opposite. In contrast "==" compares if two lists have the same value.

```
:let alist = [1, 2, 3]
:let blist = [1, 2, 3]
:echo alist is blist
```

```
0
```

```
:echo alist == blist
```

```
1
```

Note about comparing lists: Two lists are considered equal if they have the same length and all items compare equal, as with using "==". There is one exception: When comparing a number with a string they are considered different. There is no automatic type conversion, as with using "==" on variables. Example:

```
echo 4 == "4"
```

```
1
```

```
echo [4] == ["4"]
```

```
0
```

Thus comparing Lists is more strict than comparing numbers and strings. You can compare simple values this way too by putting them in a list:

```
:let a = 5
:let b = "5"
:echo a == b
```

```
1
```

```
:echo [a] == [b]
```

```
0
```

List unpack

To unpack the items in a list to individual variables, put the variables in square brackets, like list items:

```
:let [var1, var2] = mylist
```

When the number of variables does not match the number of items in the list this produces an error. To handle any extra items from the list append ";" and a variable name:

```
:let [var1, var2; rest] = mylist
```

This works like:

```
:let var1 = mylist[0]
:let var2 = mylist[1]
:let rest = mylist[2:]
```

Except that there is no error if there are only two items. "rest" will be an empty list then.

List modification

list-modification

To change a specific item of a list use :let this way:

```
:let list[4] = "four"
:let listlist[0][3] = item
```

To change part of a list you can specify the first and last item to be modified. The value must at least have the number of items in the range:

```
:let list[3:5] = [3, 4, 5]
```

To add items to a List in-place, you can use :let+= (list-concatenation):

```
:let listA = [1, 2]
:let listA += [3, 4]
```

When two variables refer to the same List, changing one List in-place will cause the referenced List to be changed in-place:

```
:let listA = [1, 2]
:let listB = listA
:let listB += [3, 4]
:echo listA
[1, 2, 3, 4]
```

Adding and removing items from a list is done with functions. Here are a few examples:

```
:call insert(list, 'a')         " prepend item 'a'
:call insert(list, 'a', 3)      " insert item 'a' before list[3]
:call add(list, "new")          " append String item
:call add(list, [1, 2])         " append a List as one new item
:call extend(list, [1, 2])      " extend the list with two more items
:let i = remove(list, 3)        " remove item 3
:unlet list[3]                  " idem
:let l = remove(list, 3, -1)    " remove items 3 to last item
:unlet list[3 : ]               " idem
:call filter(list, 'v:val !~ "x"')  " remove items with an 'x'
```

Changing the order of items in a list:

```
:call sort(list)                " sort a list alphabetically
:call reverse(list)             " reverse the order of items
:call uniq(sort(list))          " sort and remove duplicates
```

For loop

The :for loop executes commands for each item in a List, String or Blob.
A variable is set to each item in sequence. Example with a List:

```
:for item in mylist
:   call Doit(item)
:endfor
```

This works like:

```
:let index = 0
:while index < len(mylist)
:   let item = mylist[index]
:   :call Doit(item)
:   let index = index + 1
:endwhile
```

If all you want to do is modify each item in the list then the map()
function will be a simpler method than a for loop.

Just like the :let command, :for also accepts a list of variables. This
requires the argument to be a List of Lists.

```
:for [lnum, col] in [[1, 3], [2, 8], [3, 0]]
:   call Doit(lnum, col)
:endfor
```

This works like a :let command is done for each list item. Again, the types
must remain the same to avoid an error.

It is also possible to put remaining items in a List variable:

```
:for [i, j; rest] in listlist
:   call Doit(i, j)
:   if !empty(rest)
:      echo "remainder: " .. string(rest)
:   endif
:endfor
```

For a Blob one byte at a time is used.

For a String one character, including any composing characters, is used as a
String. Example:

```
for c in text
  echo 'This character is ' .. c
endfor
```

List functions

E714

Functions that are useful with a List:

```
:let r = call(funcname, list)   " call a function with an argument list
:if empty(list)                 " check if list is empty
:let l = len(list)              " number of items in list
:let big = max(list)            " maximum value in list
:let small = min(list)          " minimum value in list
:let xs = count(list, 'x')      " count nr of times 'x' appears in list
:let i = index(list, 'x')       " index of first 'x' in list
:let lines = getline(1, 10)     " get ten text lines from buffer
:call append('$', lines)        " append text lines in buffer
:let list = split("a b c")      " create list from items in a string
:let string = join(list, ', ')  " create string from list items
:let s = string(list)           " String representation of list
:call map(list, '">> " .. v:val')  " prepend ">> " to each item
```

Don't forget that a combination of features can make things simple. For example, to add up all the numbers in a list:

```
:exe 'let sum = ' .. join(nrlist, '+')
```

1.4 Dictionaries

Dict dict Dictionaries Dictionary

A Dictionary is an associative array: Each entry has a key and a value. The entry can be located with the key. The entries are stored without a specific ordering.

Dictionary creation

E720 E721 E722 E723

A Dictionary is created with a comma-separated list of entries in curly braces. Each entry has a key and a value, separated by a colon. Each key can only appear once. Examples:

```
:let mydict = {1: 'one', 2: 'two', 3: 'three'}
:let emptydict = {}
```

E713 E716 E717

A key is always a String. You can use a Number, it will be converted to a String automatically. Thus the String '4' and the number 4 will find the same entry. Note that the String '04' and the Number 04 are different, since the Number will be converted to the String '4', leading zeros are dropped. The empty string can also be used as a key.

literal-Dict #{}

To avoid having to put quotes around every key the #{} form can be used. This does require the key to consist only of ASCII letters, digits, '-' and '_'. Example:

```
:let mydict = #{zero: 0, one_key: 1, two-key: 2, 333: 3}
```

Note that 333 here is the string "333". Empty keys are not possible with #{}.

A value can be any expression. Using a Dictionary for a value creates a nested Dictionary:

```
:let nestdict = {1: {11: 'a', 12: 'b'}, 2: {21: 'c'}}
```

An extra comma after the last entry is ignored.

Accessing entries

The normal way to access an entry is by putting the key in square brackets:

```
:let val = mydict["one"]
:let mydict["four"] = 4
```

You can add new entries to an existing Dictionary this way, unlike Lists.

For keys that consist entirely of letters, digits and underscore the following form can be used expr-entry:

```
:let val = mydict.one
:let mydict.four = 4
```

Since an entry can be any type, also a List and a Dictionary, the indexing and key lookup can be repeated:

```
:echo dict.key[idx].key
```

Dictionary to List conversion

You may want to loop over the entries in a dictionary. For this you need to turn the Dictionary into a List and pass it to :for.

Most often you want to loop over the keys, using the keys() function:

```
:for key in keys(mydict)
:   echo key .. ': ' .. mydict[key]
:endfor
```

The List of keys is unsorted. You may want to sort them first:

```
:for key in sort(keys(mydict))
```

To loop over the values use the values() function:

```
:for v in values(mydict)
:   echo "value: " .. v
:endfor
```

If you want both the key and the value use the items() function. It returns

a List in which each item is a List with two items, the key and the value:

```
:for [key, value] in items(mydict)
:   echo key .. ': ' .. value
:endfor
```

Dictionary identity

dict-identity

Just like Lists you need to use copy() and deepcopy() to make a copy of a Dictionary. Otherwise, assignment results in referring to the same Dictionary:

```
:let onedict = {'a': 1, 'b': 2}
:let adict = onedict
:let adict['a'] = 11
:echo onedict['a']
11
```

Two Dictionaries compare equal if all the key-value pairs compare equal. For more info see list-identity.

Dictionary modification

dict-modification

To change an already existing entry of a Dictionary, or to add a new entry, use :let this way:

```
:let dict[4] = "four"
:let dict['one'] = item
```

Removing an entry from a Dictionary is done with remove() or :unlet. Three ways to remove the entry with key "aaa" from dict:

```
:let i = remove(dict, 'aaa')
:unlet dict.aaa
:unlet dict['aaa']
```

Merging a Dictionary with another is done with extend():

```
:call extend(adict, bdict)
```

This extends adict with all entries from bdict. Duplicate keys cause entries in adict to be overwritten. An optional third argument can change this. Note that the order of entries in a Dictionary is irrelevant, thus don't expect ":echo adict" to show the items from bdict after the older entries in adict.

Weeding out entries from a Dictionary can be done with filter():

```
:call filter(dict, 'v:val =~ "x"')
```

This removes all entries from "dict" with a value not matching 'x'. This can also be used to remove all entries:

```
call filter(dict, 0)
```

Dictionary function

Dictionary-function self E725 E862

When a function is defined with the "dict" attribute it can be used in a special way with a dictionary. Example:

```
:function Mylen() dict
:   return len(self.data)
:endfunction
:let mydict = {'data': [0, 1, 2, 3], 'len': function("Mylen")}
:echo mydict.len()
```

This is like a method in object oriented programming. The entry in the Dictionary is a Funcref. The local variable "self" refers to the dictionary the function was invoked from.

It is also possible to add a function without the "dict" attribute as a Funcref to a Dictionary, but the "self" variable is not available then.

numbered-function anonymous-function

To avoid the extra name for the function it can be defined and directly assigned to a Dictionary in this way:

```
:let mydict = {'data': [0, 1, 2, 3]}
:function mydict.len()
:   return len(self.data)
:endfunction
:echo mydict.len()
```

The function will then get a number and the value of dict.len is a Funcref
that references this function. The function can only be used through a
Funcref. It will automatically be deleted when there is no Funcref
remaining that refers to it.

It is not necessary to use the "dict" attribute for a numbered function.

If you get an error for a numbered function, you can find out what it is with
a trick. Assuming the function is 42, the command is:

```
:function g:42
```

Functions for Dictionaries

E715

Functions that can be used with a Dictionary:

```
:if has_key(dict, 'foo')        " TRUE if dict has entry with key "foo"
:if empty(dict)                 " TRUE if dict is empty
:let l = len(dict)              " number of items in dict
:let big = max(dict)            " maximum value in dict
:let small = min(dict)          " minimum value in dict
:let xs = count(dict, 'x')      " count nr of times 'x' appears in dict
:let s = string(dict)           " String representation of dict
:call map(dict, '">> " .. v:val')  " prepend ">> " to each item
```

1.5 Blobs

blob Blob Blobs E978

A Blob is a binary object. It can be used to read an image from a file and
send it over a channel, for example.

A Blob mostly behaves like a List of numbers, where each number has the
value of an 8-bit byte, from 0 to 255.

Blob creation

A Blob can be created with a blob-literal:

```
:let b = 0zFF00ED015DAF
```

Dots can be inserted between bytes (pair of hex characters) for readability,
they don't change the value:

```
:let b = 0zFF00.ED01.5DAF
```

A blob can be read from a file with readfile() passing the {type} argument
set to "B", for example:

```
:let b = readfile('image.png', 'B')
```

Blob index

blob-index E979

A byte in the Blob can be accessed by putting the index in square brackets
after the Blob. Indexes are zero-based, thus the first byte has index zero.

```
:let myblob = 0z00112233
:let byte = myblob[0]           " get the first byte: 0x00
:let byte = myblob[2]           " get the third byte: 0x22
```

A negative index is counted from the end. Index -1 refers to the last byte in
the Blob, -2 to the last but one byte, etc.

```
:let last = myblob[-1]          " get the last byte: 0x33
```

To avoid an error for an invalid index use the get() function. When an item
is not available it returns -1 or the default value you specify:

```
:echo get(myblob, idx)
:echo get(myblob, idx, 999)
```

Blob iteration

The :for loop executes commands for each byte of a Blob. The loop variable is
set to each byte in the Blob. Example:

```
:for byte in 0z112233
:   call Doit(byte)
:endfor
```

This calls Doit() with 0x11, 0x22 and 0x33.

Blob concatenation

blob-concatenation

Two blobs can be concatenated with the "+" operator:

```
:let longblob = myblob + 0z4455
:let longblob = 0z4455 + myblob
```

A blob can be concatenated with another one in-place using :let+=:

```
:let myblob += 0z6677
```

See blob-modification below for more about changing a blob in-place.

Part of a blob

A part of the Blob can be obtained by specifying the first and last index,
separated by a colon in square brackets:

```
:let myblob = 0z00112233
:let shortblob = myblob[1:2]    " get 0z1122
:let shortblob = myblob[2:-1]   " get 0z2233
```

Omitting the first index is similar to zero. Omitting the last index is
similar to -1.

```
:let endblob = myblob[2:]       " from item 2 to the end: 0z2233
:let shortblob = myblob[2:2]    " Blob with one byte: 0z22
:let otherblob = myblob[:]      " make a copy of the Blob
```

If the first index is beyond the last byte of the Blob or the second index is
before the first index, the result is an empty Blob. There is no error
message.

If the second index is equal to or greater than the length of the Blob the
length minus one is used:

```
:echo myblob[2:8]               " result: 0z2233
```

Blob modification

blob-modification

To change a specific byte of a blob use :let this way:

```
:let blob[4] = 0x44
```

When the index is just one beyond the end of the Blob, it is appended. Any
higher index is an error.

To change a sequence of bytes the [:] notation can be used:

```
let blob[1:3] = 0z445566
```

The length of the replaced bytes must be exactly the same as the value
provided. E972

To change part of a blob you can specify the first and last byte to be
modified. The value must have the same number of bytes in the range:

```
:let blob[3:5] = 0z334455
```

To add items to a Blob in-place, you can use :let+= (blob-concatenation):

```
:let blobA = 0z1122
:let blobA += 0z3344
```

When two variables refer to the same Blob, changing one Blob in-place will
cause the referenced Blob to be changed in-place:

```
:let blobA = 0z1122
:let blobB = blobA
:let blobB += 0z3344
:echo blobA
0z11223344
```

You can also use the functions add(), remove() and insert().

Blob identity

Blobs can be compared for equality:

```
if blob == 0z001122
```

And for equal identity:

```
if blob is otherblob
```

blob-identity E977

When variable "aa" is a Blob and you assign it to another variable "bb", both
variables refer to the same Blob. Then the "is" operator returns true.

When making a copy using [:] or copy() the values are the same, but the identity is different:

```
:let blob = 0z112233
:let blob2 = blob
:echo blob == blob2
```

```
1
```

```
:echo blob is blob2
```

```
1
```

```
:let blob3 = blob[:]
:echo blob == blob3
```

```
1
```

```
:echo blob is blob3
```

```
0
```

Making a copy of a Blob is done with the copy() function. Using [:] also works, as explained above.

1.6 More about variables

more-variables

If you need to know the type of a variable or expression, use the type() function.

When the '!' flag is included in the 'shada' option, global variables that start with an uppercase letter, and don't contain a lowercase letter, are stored in the shada file shada-file.

When the 'sessionoptions' option contains "global", global variables that start with an uppercase letter and contain at least one lowercase letter are stored in the session file session-file.

variable name	can be stored where
my_var_6	not
My_Var_6	session file
MY_VAR_6	shada file

It's possible to form a variable name with curly braces, see curly-braces-names.

2. Expression syntax

expression-syntax

Expression syntax summary, from least to most significant:

```
expr1   expr2
        expr2 ? expr1 : expr1   if-then-else

expr2   expr3
        expr3 || expr3 ...      logical OR

expr3   expr4
        expr4 && expr4 ...      logical AND

expr4   expr5
        expr5 == expr5          equal
        expr5 != expr5          not equal
        expr5 >  expr5          greater than
        expr5 >= expr5          greater than or equal
        expr5 <  expr5          smaller than
        expr5 <= expr5          smaller than or equal
        expr5 =~ expr5          regexp matches
        expr5 !~ expr5          regexp doesn't match

        expr5 ==? expr5         equal, ignoring case
        expr5 ==# expr5         equal, match case
        etc.                    As above, append ? for ignoring case, # for
                                matching case

        expr5 is expr5          same List, Dictionary or Blob instance
        expr5 isnot expr5       different List, Dictionary or Blob
                                instance

expr5   expr6
        expr6 +  expr6 ...      number addition, list or blob concatenation
        expr6 -  expr6 ...      number subtraction
        expr6 .  expr6 ...      string concatenation
        expr6 .. expr6 ...      string concatenation

expr6   expr7
        expr7 *  expr7 ...      number multiplication
        expr7 /  expr7 ...      number division
        expr7 %  expr7 ...      number modulo

expr7   expr8
        ! expr7                 logical NOT
```

- expr7 unary minus

+ expr7 unary plus

expr8	expr9	
	expr8[expr1]	byte of a String or item of a List
	expr8[expr1 : expr1]	substring of a String or sublist of a List
	expr8.name	entry in a Dictionary
	expr8(expr1, ...)	function call with Funcref variable
	expr8->name(expr1, ...)	method call
expr9	number	number constant
	"string"	string constant, backslash is special
	'string'	string constant, ' is doubled
	[expr1, ...]	List
	{expr1: expr1, ...}	Dictionary
	#{key: expr1, ...}	Dictionary
	&option	option value
	(expr1)	nested expression
	variable	internal variable
	va{ria}ble	internal variable with curly braces
	$VAR	environment variable
	@r	contents of register "r"
	function(expr1, ...)	function call
	func{ti}on(expr1, ...)	function call with curly braces
	{args -> expr1}	lambda expression

"..." indicates that the operations in this level can be concatenated.
Example:

```
&nu || &list && &shell == "csh"
```

All expressions within one level are parsed from left to right.

Expression nesting is limited to 1000 levels deep (300 when build with MSVC)
to avoid running out of stack and crashing. E1169

expr1 — expr1 ternary falsy-operator ?? E109

The ternary operator: expr2 ? expr1 : expr1
The falsy operator: expr2 ?? expr1

Ternary operator

The expression before the '?' is evaluated to a number. If it evaluates to
TRUE, the result is the value of the expression between the '?' and ':',
otherwise the result is the value of the expression after the ':'.
Example:

```
:echo lnum == 1 ? "top" : lnum
```

Since the first expression is an "expr2", it cannot contain another ?:. The
other two expressions can, thus allow for recursive use of ?:.
Example:

```
:echo lnum == 1 ? "top" : lnum == 1000 ? "last" : lnum
```

To keep this readable, using line-continuation is suggested:

```
:echo lnum == 1
:\      ? "top"
:\      : lnum == 1000
:\              ? "last"
:\              : lnum
```

You should always put a space before the ':', otherwise it can be mistaken for
use in a variable such as "a:1".

Falsy operator

This is also known as the "null coalescing operator", but that's too
complicated, thus we just call it the falsy operator.

The expression before the '??' is evaluated. If it evaluates to
truthy, this is used as the result. Otherwise the expression after the '??'
is evaluated and used as the result. This is most useful to have a default
value for an expression that may result in zero or empty:

```
echo theList ?? 'list is empty'
echo GetName() ?? 'unknown'
```

These are similar, but not equal:

```
expr2 ?? expr1
expr2 ? expr2 : expr1
```

In the second line "expr2" is evaluated twice.

expr2 and expr3 — expr2 expr3

expr3 || expr3 ..   logical OR — expr-barbar

expr4 && expr4 ..   logical AND — expr-&&

The "||" and "&&" operators take one argument on each side. The arguments are (converted to) Numbers. The result is:

input	output		
n1 n2 n1 || n2 n1 && n2			
FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE
TRUE	FALSE	TRUE	FALSE
TRUE	TRUE	TRUE	TRUE

The operators can be concatenated, for example:

```
&nu || &list && &shell == "csh"
```

Note that "&&" takes precedence over "||", so this has the meaning of:

```
&nu || (&list && &shell == "csh")
```

Once the result is known, the expression "short-circuits", that is, further arguments are not evaluated. This is like what happens in C. For example:

```
let a = 1
echo a || b
```

This is valid even if there is no variable called "b" because "a" is TRUE, so the result must be TRUE. Similarly below:

```
echo exists("b") && b == "yes"
```

This is valid whether "b" has been defined or not. The second clause will only be evaluated if "b" has been defined.

expr4 **expr4**

expr5 {cmp} expr5

Compare two expr5 expressions, resulting in a 0 if it evaluates to false, or 1 if it evaluates to true.

expr-== expr-!= expr-> expr->=

expr-< expr-<= expr-=~ expr-!~
expr-==# expr-!=# expr-># expr->=#
expr-<# expr-<=# expr-=~# expr-!~#
expr-==? expr-!=? expr->? expr->=?
expr-<? expr-<=? expr-=~? expr-!~?
expr-is expr-isnot expr-is# expr-isnot#
expr-is? expr-isnot?

use 'ignorecase'	match case	ignore case	
equal	==	==#	==?
not equal	!=	!=#	!=?
greater than	>	>#	>?
greater than or equal	>=	>=#	>=?
smaller than	<	<#	<?
smaller than or equal	<=	<=#	<=?
regexp matches	=~	=~#	=~?
regexp doesn't match	!~	!~#	!~?
same instance	is	is#	is?
different instance	isnot	isnot#	isnot?

Examples:
"abc" ==# "Abc" evaluates to 0
"abc" ==? "Abc" evaluates to 1
"abc" == "Abc" evaluates to 1 if 'ignorecase' is set, 0 otherwise

E691 E692

A List can only be compared with a List and only "equal", "not equal", "is" and "isnot" can be used. This compares the values of the list, recursively. Ignoring case means case is ignored when comparing item values.

E735 E736

A Dictionary can only be compared with a Dictionary and only "equal", "not equal", "is" and "isnot" can be used. This compares the key/values of the Dictionary recursively. Ignoring case means case is ignored when comparing item values.

E694

A Funcref can only be compared with a Funcref and only "equal", "not equal", "is" and "isnot" can be used. Case is never ignored. Whether arguments or a Dictionary are bound (with a partial) matters. The Dictionaries must also be equal (or the same, in case of "is") and the arguments must be equal (or the same).

To compare Funcrefs to see if they refer to the same function, ignoring bound Dictionary and arguments, use get() to get the function name:

```
if get(Part1, 'name') == get(Part2, 'name')
   " Part1 and Part2 refer to the same function
```

Using "is" or "isnot" with a List, Dictionary or Blob checks whether
the expressions are referring to the same List, Dictionary or Blob
instance. A copy of a List is different from the original List. When
using "is" without a List, Dictionary or Blob, it is equivalent to
using "equal", using "isnot" is equivalent to using "not equal". Except that
a different type means the values are different:

```
echo 4 == '4'
1
echo 4 is '4'
0
echo 0 is []
0
```

"is#"/"isnot#" and "is?"/"isnot?" can be used to match and ignore case.

When comparing a String with a Number, the String is converted to a Number,
and the comparison is done on Numbers. This means that:

```
echo 0 == 'x'
1
```

because 'x' converted to a Number is zero. However:

```
echo [0] == ['x']
0
```

Inside a List or Dictionary this conversion is not used.

When comparing two Strings, this is done with strcmp() or stricmp(). This
results in the mathematical difference (comparing byte values), not
necessarily the alphabetical difference in the local language.

When using the operators with a trailing '#', or the short version and
'ignorecase' is off, the comparing is done with strcmp(): case matters.

When using the operators with a trailing '?', or the short version and
'ignorecase' is set, the comparing is done with stricmp(): case is ignored.

'smartcase' is not used.

The "=~" and "!~" operators match the lefthand argument with the righthand
argument, which is used as a pattern. See pattern for what a pattern is.
This matching is always done like 'magic' was set and 'cpoptions' is empty, no
matter what the actual value of 'magic' or 'cpoptions' is. This makes scripts
portable. To avoid backslashes in the regexp pattern to be doubled, use a
single-quote string, see literal-string.
Since a string is considered to be a single line, a multi-line pattern
(containing \n, backslash-n) will not match. However, a literal NL character
can be matched like an ordinary character. Examples:
```
        "foo\nbar" =~ "\n"      evaluates to 1
        "foo\nbar" =~ "\\n"     evaluates to 0
```

expr5 and expr6 **expr5** **expr6**

expr6 + expr6   Number addition, List or Blob concatenation   expr-+
expr6 - expr6   Number subtraction   expr--

expr6 . expr6   String concatenation   expr-.

expr6 .. expr6 String concatenation   expr-..

For Lists only "+" is possible and then both expr6 must be a list. The
result is a new list with the two lists Concatenated.

For String concatenation ".." is preferred, since "." is ambiguous, it is also
used for Dict member access and floating point numbers.

expr7 * expr7 Number multiplication   expr-star

expr7 / expr7 Number division   expr-/

expr7 % expr7 Number modulo   expr-%

For all, except "." and "..", Strings are converted to Numbers.
For bitwise operators see and(), or() and xor().

Note the difference between "+" and ".":
```
        "123" + "456" = 579
        "123" . "456" = "123456"
```

Since '.' has the same precedence as '+' and '-', you need to read:

```
1 . 90 + 90.0
```

As:

```
(1 . 90) + 90.0
```

That works, since the String "190" is automatically converted to the Number
190, which can be added to the Float 90.0. However:

```
1 . 90 * 90.0
```

Should be read as:

```
1 . (90 * 90.0)
```

Since '.' has lower precedence than "*". This does NOT work, since this
attempts to concatenate a Float and a String.

When dividing a Number by zero the result depends on the value:
```
  0 / 0  = -0x80000000  (like NaN for Float)
 >0 / 0  =  0x7fffffff  (like positive infinity)
 <0 / 0  = -0x7fffffff  (like negative infinity)
(before Vim 7.2 it was always 0x7fffffff)
```

When 64-bit Number support is enabled:
```
  0 / 0  = -0x8000000000000000  (like NaN for Float)
 >0 / 0  =  0x7fffffffffffffff  (like positive infinity)
 <0 / 0  = -0x7fffffffffffffff  (like negative infinity)
```

When the righthand side of '%' is zero, the result is 0.

None of these work for Funcrefs.

. and % do not work for Float. E804

expr7 expr7

! expr7 logical NOT expr-!

- expr7 unary minus expr-unary--

+ expr7 unary plus expr-unary-+

For '!' TRUE becomes FALSE, FALSE becomes TRUE (one).
For '-' the sign of the number is changed.
For '+' the number is unchanged. **Note:** "++" has no effect.

A String will be converted to a Number first.

These three can be repeated and mixed. Examples:
```
        !-1         == 0
        !!8         == 1
        --9         == 9
```

expr8 expr8

This expression is either expr9 or a sequence of the alternatives below,
in any order. E.g., these are all possible:
```
        expr8[expr1].name
        expr8.name[expr1]
        expr8(expr1, ...)[expr1].name
        expr8->(expr1, ...)[expr1]
```
Evaluation is always from left to right.

expr8[expr1] item of String or List expr-[] E111 subscript

In legacy Vim script:
If expr8 is a Number or String this results in a String that contains the
expr1'th single byte from expr8. expr8 is used as a String (a number is
automatically converted to a String), expr1 as a Number. This doesn't
recognize multibyte encodings, see byteidx() for an alternative, or use
split() to turn the string into a list of characters. Example, to get the
byte under the cursor:

```
:let c = getline(".")[col(".") - 1]
```

Index zero gives the first byte. This is like it works in C. Careful:
text column numbers start with one! Example, to get the byte under the
cursor:

```
:let c = getline(".")[col(".") - 1]
```

Index zero gives the first byte. Careful: text column numbers start with one!

If the length of the String is less than the index, the result is an empty
String. A negative index always results in an empty string (reason: backward
compatibility). Use [-1:] to get the last byte.

If expr8 is a List then it results the item at index expr1. See list-index
for possible index values. If the index is out of range this results in an
error. Example:

```
:let item = mylist[-1]          " get last item
```

Generally, if a List index is equal to or higher than the length of the List, or more negative than the length of the List, this results in an error.

expr8[expr1a : expr1b] substring or sublist expr-[:] substring

If expr8 is a String this results in the substring with the bytes or characters from expr1a to and including expr1b. expr8 is used as a String, expr1a and expr1b are used as a Number.

In legacy Vim script the indexes are byte indexes. This doesn't recognize multibyte encodings, see byteidx() for computing the indexes. If expr8 is a Number it is first converted to a String.

The item at index expr1b is included, it is inclusive. For an exclusive index use the slice() function.

If expr1a is omitted zero is used. If expr1b is omitted the length of the string minus one is used.

A negative number can be used to measure from the end of the string. -1 is the last character, -2 the last but one, etc.

If an index goes out of range for the string characters are omitted. If expr1b is smaller than expr1a the result is an empty string.

Examples:

```
:let c = name[-1:]              " last byte of a string
:let c = name[0:-1]             " the whole string
:let c = name[-2:-2]            " last but one byte of a string
:let s = line(".")[4:]          " from the fifth byte to the end
:let s = s[:-3]                 " remove last two bytes
```

slice

If expr8 is a List this results in a new List with the items indicated by the indexes expr1a and expr1b. This works like with a String, as explained just above. Also see sublist below. Examples:

```
:let l = mylist[:3]             " first four items
:let l = mylist[4:4]            " List with one item
:let l = mylist[:]              " shallow copy of a List
```

If expr8 is a Blob this results in a new Blob with the bytes in the indexes expr1a and expr1b, inclusive. Examples:

```
:let b = 0zDEADBEEF
:let bs = b[1:2]                " 0zADBE
:let bs = b[]                   " copy of 0zDEADBEEF
```

Using expr8[expr1] or expr8[expr1a : expr1b] on a Funcref results in an error.

Watch out for confusion between a namespace and a variable followed by a colon for a sublist:

```
mylist[n:]     " uses variable n
mylist[s:]     " uses namespace s:, error!
```

expr8.name entry in a Dictionary expr-entry

If expr8 is a Dictionary and it is followed by a dot, then the following name will be used as a key in the Dictionary. This is just like: expr8[name].

The name must consist of alphanumeric characters, just like a variable name, but it may start with a number. Curly braces cannot be used.

There must not be white space before or after the dot.

Examples:

```
:let dict = {"one": 1, 2: "two"}
:echo dict.one          " shows "1"
:echo dict.2            " shows "two"
:echo dict .2           " error because of space before the dot
```

Note that the dot is also used for String concatenation. To avoid confusion always put spaces around the dot for String concatenation.

expr8(expr1, ...) Funcref function call E1085

When expr8 is a Funcref type variable, invoke the function it refers to.

expr8->name([args]) method call method ->
expr8->{lambda}([args])

E260 E276

For methods that are also available as global functions this is the same as:

```
name(expr8 [, args])
```

There can also be methods specifically for the type of "expr8".

This allows for chaining, passing the value that one method returns to the
next method:

```
mylist->filter(filterexpr)->map(mapexpr)->sort()->join()
```

Example of using a lambda:

```
GetPercentage()->{x -> x * 100}()->printf('%d%%')
```

When using -> the expr7 operators will be applied first, thus:

```
-1.234->string()
```

Is equivalent to:

```
(-1.234)->string()
```

And NOT:

```
-(1.234->string())
```

E274

"->name(" must not contain white space. There can be white space before the
"->" and after the "(", thus you can split the lines like this:

```
mylist
\ ->filter(filterexpr)
\ ->map(mapexpr)
\ ->sort()
\ ->join()
```

When using the lambda form there must be no white space between the } and the

(.

expr9

number

number number constant expr-number

0x hex-number 0o octal-number 0b binary-number

Integer numbers can be written in multiple forms:

{N}	decimal
0x{N} or 0X{N}	hexadecimal
0o{N} or 0O{N}	octal
0{N}	octal
0b{N} or 0B{N}	binary

{N} is a sequence of decimal digits (0-9), hexadecimal digits (0-9, a-f and
A-F), octal digits (0-7) or binary digits (0 or 1).

A leading "+" or "-" is not part of an integer literal. E.g., -123 is an
expression, the unary operator expr-unary-- applied to the number 123.

Examples:

```
0xDEAD
57005
0b1101111010101101
0o157255
```

floating-point-format

Floating point numbers can be written in two forms:

```
{N}.{M}
{N}.{M}[eE][-+]{exp}
```

{N} and {M} are sequences of decimal digits Both {N} and {M} must be present.
{exp} is the exponent, power of 10.
Only a decimal point is accepted, not a comma. No matter what the current
locale is.

A leading "+" or "-" is not part of a floating point literal. E.g., -1.0e-3
is an expression, the unary operator expr-unary-- applied to the float 1.0e-3.
The "-" in the exponent is part of the literal.

Examples:

```
123.456
0.0001
55.0
0.123
1.234e03
1.0E-6
3.1416e+88
```

These are INVALID:

```
3.              empty {M}
```

```
        1e40            missing .{M}
```

Rationale:
Before floating point was introduced, the text "123.456" was interpreted as the two numbers "123" and "456", both converted to a string and concatenated, resulting in the string "123456". Since this was considered pointless, and we could not find it intentionally being used in Vim scripts, this backwards incompatibility was accepted in favor of being able to use the normal notation for floating point numbers.

float-pi float-e

A few useful values to copy&paste:

```
:let pi = 3.14159265359
:let e  = 2.71828182846
```

Or, if you don't want to write them in as floating-point literals, you can also use functions, like the following:

```
:let pi = acos(-1.0)
:let e  = exp(1.0)
```

floating-point-precision

The precision and range of floating points numbers depends on what "double" means in the library Vim was compiled with. There is no way to change this at runtime.

The default for displaying a Float is to use 6 decimal places, like using printf("%g", f). You can select something else when using the printf() function. Example:

```
:echo printf('%.15e', atan(1))
```

```
        7.853981633974483e-01
```

string

string String expr-string E114

```
"string"                string constant
```

expr-quote

Note that double quotes are used.

A string constant accepts these special characters:

```
\...    three-digit octal number (e.g., "\316")
\..     two-digit octal number (must be followed by non-digit)
\.      one-digit octal number (must be followed by non-digit)
\x..    byte specified with two hex numbers (e.g., "\x1f")
\x.     byte specified with one hex number (must be followed by non-hex char)
\X..    same as \x..
\X.     same as \x.
\u....  character specified with up to 4 hex numbers, stored as UTF-8
        (e.g., "\u02a4")
\U....  same as \u but allows up to 8 hex numbers.
\b      backspace <BS>
\e      escape <Esc>
\f      formfeed 0x0C
\n      newline <NL>
\r      return <CR>
\t      tab <Tab>
\\      backslash
\"      double quote
\<xxx>  Special key named "xxx".  e.g. "\<C-W>" for CTRL-W.  This is for use
        in mappings, the 0x80 byte is escaped.
        To use the double quote character it must be escaped: "<M-\">".
        Don't use <Char-xxxx> to get a UTF-8 character, use \uxxxx as
        mentioned above.
\<*xxx> Like \<xxx> but prepends a modifier instead of including it in the
        character.  E.g. "\<C-w>" is one character 0x17 while "\<*C-w>" is
        four bytes: 3 for the CTRL modifier and then character "W".
```

Note that "\xff" is stored as the byte 255, which may be invalid in some encodings. Use "\u00ff" to store character 255 correctly as UTF-8.

Note that "\000" and "\x00" force the end of the string.

blob-literal

blob-literal E973

Hexadecimal starting with 0z or 0Z, with an arbitrary number of bytes.
The sequence must be an even number of hex characters. Example:

```
:let b = 0zFF00ED015DAF
```

literal-string

literal-string E115

```
'string'                string constant
```

expr-'

Note that single quotes are used.

This string is taken as it is. No backslashes are removed or have a special meaning. The only exception is that two quotes stand for one quote.

Single quoted strings are useful for patterns, so that backslashes do not need to be doubled. These two commands are equivalent:

```
if a =~ "\\s*"
if a =~ '\s*'
```

interpolated-string

$quote **interpolated-string**

```
$"string"               interpolated string constant                    expr-$quote

$'string'               interpolated literal string constant    expr-$'
```

Interpolated strings are an extension of the string and literal-string, allowing the inclusion of Vim script expressions (see expr1). Any expression returning a value can be enclosed between curly braces. The value is converted to a string. All the text and results of the expressions are concatenated to make a new string.

E1278 E1279

To include an opening brace '{' or closing brace '}' in the string content double it. For double quoted strings using a backslash also works. A single closing brace '}' will result in an error.

Examples:

```
let your_name = input("What's your name? ")
```

What's your name? Peter

```
echo
echo $"Hello, {your_name}!"
```

Hello, Peter!

```
echo $"The square root of {{9}} is {sqrt(9)}"
```

The square root of {9} is 3.0

string-offset-encoding

A string consists of multiple characters. UTF-8 uses one byte for ASCII characters, two bytes for other latin characters and more bytes for other characters.

A string offset can count characters or bytes. Other programs may use UTF-16 encoding (16-bit words) and an offset of UTF-16 words. Some functions use byte offsets, usually for UTF-8 encoding. Other functions use character offsets, in which case the encoding doesn't matter.

The different offsets for the string "a©😊" are below:

```
  UTF-8 offsets:
      [0]: 61, [1]: C2, [2]: A9, [3]: F0, [4]: 9F, [5]: 98, [6]: 8A
  UTF-16 offsets:
      [0]: 0061, [1]: 00A9, [2]: D83D, [3]: DE0A
  UTF-32 (character) offsets:
      [0]: 00000061, [1]: 000000A9, [2]: 0001F60A
```

You can use the "g8" and "ga" commands on a character to see the decimal/hex/octal values.

The functions byteidx(), utf16idx() and charidx() can be used to convert between these indices. The functions strlen(), strutf16len() and strcharlen() return the number of bytes, UTF-16 code units and characters in a string respectively.

option

expr-option **E112** **E113**

```
&option                 option value, local value if possible
&g:option               global option value
&l:option               local option value
```

Examples:

```
echo "tabstop is " .. &tabstop
if &expandtab
```

Any option name can be used here. See options. When using the local value and there is no buffer-local or window-local value, the global value is used anyway.

register

expr-register **@r**

```
@r                      contents of register 'r'
```

The result is the contents of the named register, as a single string. Newlines are inserted where required. To get the contents of the unnamed register use @" or @@. See registers for an explanation of the available registers.

When using the '=' register you get the expression itself, not what it evaluates to. Use eval() to evaluate it.

```
nesting                                                 expr-nesting E110
-------
(expr1)                 nested expression
```

environment variable

expr-env

$VAR environment variable

The String value of any environment variable. When it is not defined, the result is an empty string.

The functions getenv() and setenv() can also be used and work for environment variables with non-alphanumeric names.
The function environ() can be used to get a Dict with all environment variables.

expr-env-expand

Note that there is a difference between using $VAR directly and using expand("$VAR"). Using it directly will only expand environment variables that are known inside the current Vim session. Using expand() will first try using the environment variables known inside the current Vim session. If that fails, a shell will be used to expand the variable. This can be slow, but it does expand all variables that the shell knows about. Example:

```
:echo $shell
:echo expand("$shell")
```

The first one probably doesn't echo anything, the second echoes the $shell variable (if your shell supports it).

internal variable

expr-variable

variable internal variable
See below internal-variables.

function call

expr-function E116 E118 E119 E120

function(expr1, ...) function call
See below functions.

lambda expression

expr-lambda lambda

{args -> expr1} lambda expression E451

A lambda expression creates a new unnamed function which returns the result of evaluating expr1. Lambda expressions differ from user-functions in the following ways:

1. The body of the lambda expression is an expr1 and not a sequence of Ex commands.
2. The prefix "a:" should not be used for arguments. E.g.:

```
:let F = {arg1, arg2 -> arg1 - arg2}
:echo F(5, 2)
```

```
3
```

The arguments are optional. Example:

```
:let F = {-> 'error function'}
:echo F('ignored')
```

```
error function
```

closure

Lambda expressions can access outer scope variables and arguments. This is often called a closure. Example where "i" and "a:arg" are used in a lambda while they already exist in the function scope. They remain valid even after the function returns:

```
:function Foo(arg)
:  let i = 3
:  return {x -> x + i - a:arg}
:endfunction
:let Bar = Foo(4)
:echo Bar(6)
```

```
5
```

Note that the variables must exist in the outer scope before the lambda is defined for this to work. See also :func-closure.

Lambda and closure support can be checked with:

```
if has('lambda')
```

Examples for using a lambda expression with sort(), map() and filter():

```
:echo map([1, 2, 3], {idx, val -> val + 1})
```

```
[2, 3, 4]
```

```
:echo sort([3,7,2,1,4], {a, b -> a - b})
```

```
        [1, 2, 3, 4, 7]
```

The lambda expression is also useful for jobs and timers:

```
:let timer = timer_start(500,
                \ {-> execute("echo 'Handler called'", "")},
                \ {'repeat': 3})
```

```
        Handler called
        Handler called
        Handler called
```

Note that it is possible to cause memory to be used and not freed if the
closure is referenced by the context it depends on:

```
function Function()
   let x = 0
   let F = {-> x}
 endfunction
```

The closure uses "x" from the function scope, and "F" in that same scope
refers to the closure. This cycle results in the memory not being freed.
Recommendation: don't do this.

Notice how execute() is used to execute an Ex command. That's ugly though.

Lambda expressions have internal names like '<lambda>42'. If you get an error
for a lambda expression, you can find what it is with the following command:

```
:function <lambda>42
```

See also: numbered-function

3. Internal variable internal-variables E461

An internal variable name can be made up of letters, digits and '_'. But it
cannot start with a digit. It's also possible to use curly braces, see
curly-braces-names.

An internal variable is created with the ":let" command :let.
An internal variable is explicitly destroyed with the ":unlet" command
:unlet.
Using a name that is not an internal variable or refers to a variable that has
been destroyed results in an error.

variable-scope

There are several name spaces for variables. Which one is to be used is
specified by what is prepended:

(nothing)		In a function: local to a function; otherwise: global
buffer-variable	b:	Local to the current buffer.
window-variable	w:	Local to the current window.
tabpage-variable	t:	Local to the current tab page.
global-variable	g:	Global.
local-variable	l:	Local to a function.
script-variable	s:	Local to a :sourced Vim script.
function-argument	a:	Function argument (only inside a function).
vim-variable	v:	Global, predefined by Vim.

The scope name by itself can be used as a Dictionary. For example, to
delete all script-local variables:

```
:for k in keys(s:)
:    unlet s:[k]
:endfor
```

buffer-variable b:var b:

A variable name that is preceded with "b:" is local to the current buffer.
Thus you can have several "b:foo" variables, one for each buffer.
This kind of variable is deleted when the buffer is wiped out or deleted with
:bdelete.

One local buffer variable is predefined:

b:changedtick changetick

b:changedtick The total number of changes to the current buffer. It is
incremented for each change. An undo command is also a change
in this case. Resetting 'modified' when writing the buffer is
also counted.
This can be used to perform an action only when the buffer has
changed. Example:

```
:if my_changedtick != b:changedtick
:       let my_changedtick = b:changedtick
:       call My_Update()
:endif
```

You cannot change or delete the b:changedtick variable.

window-variable w:var w:

A variable name that is preceded with "w:" is local to the current window. It
is deleted when the window is closed.

tabpage-variable t:var t:
A variable name that is preceded with "t:" is local to the current tab page,
It is deleted when the tab page is closed.

global-variable g:var g:
Inside functions global variables are accessed with "g:". Omitting this will
access a variable local to a function. But "g:" can also be used in any other
place if you like.

local-variable l:var l:
Inside functions local variables are accessed without prepending anything.
But you can also prepend "l:" if you like. However, without prepending "l:"
you may run into reserved variable names. For example "count". By itself it
refers to "v:count". Using "l:count" you can have a local variable with the
same name.

script-variable s:var
In a Vim script variables starting with "s:" can be used. They cannot be
accessed from outside of the scripts, thus are local to the script.

They can be used in:
- commands executed while the script is sourced
- functions defined in the script
- autocommands defined in the script
- functions and autocommands defined in functions and autocommands which were
  defined in the script (recursively)
- user defined commands defined in the script

Thus not in:
- other scripts sourced from this one
- mappings
- menus
- etc.

Script variables can be used to avoid conflicts with global variable names.
Take this example:

```
let s:counter = 0
function MyCounter()
  let s:counter = s:counter + 1
  echo s:counter
endfunction
command Tick call MyCounter()
```

You can now invoke "Tick" from any script, and the "s:counter" variable in
that script will not be changed, only the "s:counter" in the script where
"Tick" was defined is used.

Another example that does the same:

```
let s:counter = 0
command Tick let s:counter = s:counter + 1 | echo s:counter
```

When calling a function and invoking a user-defined command, the context for
script variables is set to the script where the function or command was
defined.

The script variables are also available when a function is defined inside a
function that is defined in a script. Example:

```
let s:counter = 0
function StartCounting(incr)
  if a:incr
    function MyCounter()
      let s:counter = s:counter + 1
    endfunction
  else
    function MyCounter()
      let s:counter = s:counter - 1
    endfunction
  endif
endfunction
```

This defines the MyCounter() function either for counting up or counting down

when calling StartCounting(). It doesn't matter from where StartCounting() is called, the s:counter variable will be accessible in MyCounter().

When the same script is sourced again it will use the same script variables. They will remain valid as long as Vim is running. This can be used to maintain a counter:

```
if !exists("s:counter")
  let s:counter = 1
  echo "script executed for the first time"
else
  let s:counter = s:counter + 1
  echo "script executed " .. s:counter .. " times now"
endif
```

Note that this means that filetype plugins don't get a different set of script variables for each buffer. Use local buffer variables instead b:var.

PREDEFINED VIM VARIABLES vim-variable

E963

The alphabetic list of all builtin variables and details are in a separate help file: vvars.

4. Builtin Functions vim-function functions

The Vimscript subsystem (referred to as "eval" internally) provides builtin functions. Scripts can also define user-functions.

See function-list to browse functions by topic.

The alphabetic list of all builtin functions and details are in a separate help file: vimscript-functions.

5. Defining functions user-function

New functions can be defined. These can be called just like builtin functions. The function takes arguments, executes a sequence of Ex commands and can return a value.

You can find most information about defining functions in userfunc.txt.

6. Curly braces names curly-braces-names

In most places where you can use a variable, you can use a "curly braces name" variable. This is a regular variable name with one or more expressions wrapped in braces {} like this:

```
my_{adjective}_variable
```

When Vim encounters this, it evaluates the expression inside the braces, puts that in place of the expression, and re-interprets the whole as a variable name. So in the above example, if the variable "adjective" was set to "noisy", then the reference would be to "my_noisy_variable", whereas if "adjective" was set to "quiet", then it would be to "my_quiet_variable".

One application for this is to create a set of variables governed by an option value. For example, the statement

```
echo my_{&background}_message
```

would output the contents of "my_dark_message" or "my_light_message" depending on the current value of 'background'.

You can use multiple brace pairs:

```
echo my_{adverb}_{adjective}_message
```

..or even nest them:

```
echo my_{ad{end_of_word}}_message
```

where "end_of_word" is either "verb" or "jective".

However, the expression inside the braces must evaluate to a valid single variable name, e.g. this is invalid:

```
:let foo='a + b'
:echo c{foo}d
```

.. since the result of expansion is "ca + bd", which is not a variable name.

curly-braces-function-names

You can call and define functions by an evaluated name in a similar way. Example:

```
:let func_end='whizz'
:call my_func_{func_end}(parameter)
```

This would call the function "my_func_whizz(parameter)".

This does NOT work:

```
:let i = 3
:let @{i} = ''  " error
:echo @{i}      " error
```

7. Commands

expression-commands

:let {var-name} = {expr1} :let E18
Set internal variable {var-name} to the result of the expression {expr1}. The variable will get the type from the {expr}. If {var-name} didn't exist yet, it is created.

:let {var-name}[{idx}] = {expr1} E689

Set a list item to the result of the expression {expr1}. {var-name} must refer to a list and {idx} must be a valid index in that list. For nested list the index can be repeated.
This cannot be used to add an item to a List.
This cannot be used to set a byte in a String. You can do that like this:

```
:let var = var[0:2] .. 'X' .. var[4:]
```

When {var-name} is a Blob then {idx} can be the length of the blob, in which case one byte is appended.

E711 E719
:let {var-name}[{idx1}:{idx2}] = {expr1} E708 E709 E710
Set a sequence of items in a List to the result of the expression {expr1}, which must be a list with the correct number of items.
{idx1} can be omitted, zero is used instead.
{idx2} can be omitted, meaning the end of the list.
When the selected range of items is partly past the end of the list, items will be added.

:let+= :let-= :letstar=
:let/= :let%= :let.= :let..= E734
:let {var} += {expr1} Like ":let {var} = {var} + {expr1}".
:let {var} -= {expr1} Like ":let {var} = {var} - {expr1}".
:let {var} *= {expr1} Like ":let {var} = {var} * {expr1}".
:let {var} /= {expr1} Like ":let {var} = {var} / {expr1}".
:let {var} %= {expr1} Like ":let {var} = {var} % {expr1}".
:let {var} .= {expr1} Like ":let {var} = {var} . {expr1}".
:let {var} ..= {expr1} Like ":let {var} = {var} .. {expr1}".
These fail if {var} was not set yet and when the type of {var} and {expr1} don't fit the operator.
+= modifies a List or a Blob in-place instead of creating a new one.

:let ${env-name} = {expr1} :let-environment :let-$
Set environment variable {env-name} to the result of the expression {expr1}. The type is always String.
:let ${env-name} .= {expr1}
:let ${env-name} ..= {expr1}
Append {expr1} to the environment variable {env-name}. If the environment variable didn't exist yet this works like "=".

:let @{reg-name} = {expr1} :let-register :let-@
Write the result of the expression {expr1} in register {reg-name}. {reg-name} must be a single letter, and must be the name of a writable register (see registers). "@@" can be used for the unnamed register, "@/" for the search pattern.
If the result of {expr1} ends in a <CR> or <NL>, the register will be linewise, otherwise it will be set to charwise.
This can be used to clear the last search pattern:

```
:let @/ = ""
```

This is different from searching for an empty string, that would match everywhere.

:let @{reg-name} .= {expr1}
:let @{reg-name} ..= {expr1}
Append {expr1} to register {reg-name}. If the register was empty it's like setting it to {expr1}.

:let &{option-name} = {expr1} :let-option :let-&
Set option {option-name} to the result of the

expression {expr1}. A String or Number value is
always converted to the type of the option.
For an option local to a window or buffer the effect
is just like using the :set command: both the local
value and the global value are changed.
Example:

```
:let &path = &path .. ',/usr/local/include'
```

:let &{option-name} .= {expr1}
:let &{option-name} ..= {expr1}
For a string option: Append {expr1} to the value.
Does not insert a comma like :set+=.

:let &{option-name} += {expr1}
:let &{option-name} -= {expr1}
For a number or boolean option: Add or subtract
{expr1}.

:let &l:{option-name} = {expr1}
:let &l:{option-name} += {expr1}
:let &l:{option-name} -= {expr1}
:let &l:{option-name} .= {expr1}
:let &l:{option-name} ..= {expr1}
Like above, but only set the local value of an option
(if there is one). Works like :setlocal.

:let &g:{option-name} = {expr1}
:let &g:{option-name} += {expr1}
:let &g:{option-name} -= {expr1}
:let &g:{option-name} .= {expr1}
:let &g:{option-name} ..= {expr1}
Like above, but only set the global value of an option
(if there is one). Works like :setglobal.

:let [{name1}, {name2}, ...] = {expr1} :let-unpack E687 E688
{expr1} must evaluate to a List. The first item in
the list is assigned to {name1}, the second item to
{name2}, etc.
The number of names must match the number of items in
the List.
Each name can be one of the items of the ":let"
command as mentioned above.
Example:

```
:let [s, item] = GetItem(s)
```

Detail: {expr1} is evaluated first, then the
assignments are done in sequence. This matters if
{name2} depends on {name1}. Example:

```
:let x = [0, 1]
:let i = 0
:let [i, x[i]] = [1, 2]
:echo x
```

The result is [0, 2].

:let [{name1}, {name2}, ...] += {expr1}
:let [{name1}, {name2}, ...] -= {expr1}
:let [{name1}, {name2}, ...] *= {expr1}
:let [{name1}, {name2}, ...] /= {expr1}
:let [{name1}, {name2}, ...] %= {expr1}
:let [{name1}, {name2}, ...] .= {expr1}
:let [{name1}, {name2}, ...] ..= {expr1}
Like above, but add, subtract, multiply, divide,
modulo, or append the value for each List item.

:let [{name}, ..., ; {lastname}] = {expr1} E452

Like :let-unpack above, but the List may have more
items than there are names. A list of the remaining
items is assigned to {lastname}. If there are no
remaining items {lastname} is set to an empty list.
Example:

```
:let [a, b; rest] = ["aval", "bval", 3, 4]
```

:let [{name}, ..., ; {lastname}] += {expr1}
:let [{name}, ..., ; {lastname}] -= {expr1}
:let [{name}, ..., ; {lastname}] .= {expr1}
:let [{name}, ..., ; {lastname}] ..= {expr1}
Like above, but add/subtract/append the value for each
List item.

:let=<< :let-heredoc
E990 E991 E172 E221 E1145
:let {var-name} =<< [trim] [eval] {endmarker}
text...
text...
{endmarker}
Set internal variable {var-name} to a List

containing the lines of text bounded by the string
{endmarker}.

If "eval" is not specified, then each line of text is
used as a literal-string, except that single quotes
does not need to be doubled.
If "eval" is specified, then any Vim expression in the
form {expr} is evaluated and the result replaces the
expression, like with interpolated-string.
Example where $HOME is expanded:

```
let lines =<< trim eval END
  some text
  See the file {$HOME}/.vimrc
  more text
END
```

There can be multiple Vim expressions in a single line
but an expression cannot span multiple lines. If any
expression evaluation fails, then the assignment
fails.

{endmarker} must not contain white space.
{endmarker} cannot start with a lower case character.
The last line should end only with the {endmarker}
string without any other character. Watch out for
white space after {endmarker}!

Without "trim" any white space characters in the lines
of text are preserved. If "trim" is specified before
{endmarker}, then indentation is stripped so you can
do:

```
let text =<< trim END
   if ok
     echo 'done'
   endif
END
```

Results in: ["if ok", " echo 'done'", "endif"]
The end marker must line up with "let" and the
indentation of the first line is removed from all the
text lines.
Specifically: all the leading indentation exactly
matching the leading indentation of the first
non-empty text line is stripped from the input lines.
All leading indentation exactly matching the leading
indentation before let is stripped from the line
containing {endmarker}. Note that the difference
between space and tab matters here.

If {var-name} didn't exist yet, it is created.
Cannot be followed by another command, but can be
followed by a comment.

To avoid line continuation to be applied, consider
adding 'C' to 'cpoptions':

```
set cpo+=C
let var =<< END
   \ leading backslash
END
set cpo-=C
```

Examples:

```
let var1 =<< END
Sample text 1
    Sample text 2
Sample text 3
END

let data =<< trim DATA
        1 2 3 4
        5 6 7 8
DATA

let code =<< trim eval CODE
   let v = {10 + 20}
   let h = "{$HOME}"
   let s = "{Str1()} abc {Str2()}"
   let n = {MyFunc(3, 4)}
CODE
```

E121

:let {var-name} ... List the value of variable {var-name}. Multiple
variable names may be given. Special names recognized
here: E738

  g: global variables
  b: local buffer variables

```
                        w:    local window variables
                        t:    local tab page variables
                        s:    script-local variables
                        l:    local function variables
                        v:    Vim variables.
```

:let List the values of all variables. The type of the variable is indicated before the value:

```
    <nothing>   String
        #       Number
        *       Funcref
```

:unl[et][!] {name} ... :unlet :unl E108 E795

Remove the internal variable {name}. Several variable names can be given, they are all removed. The name may also be a List or Dictionary item.
With [!] no error message is given for non-existing variables.
One or more items from a List can be removed:

```
:unlet list[3]    " remove fourth item
:unlet list[3:]   " remove fourth item to last
```

One item from a Dictionary can be removed at a time:

```
:unlet dict['two']
:unlet dict.two
```

This is especially useful to clean up used global variables and script-local variables (these are not deleted when the script ends). Function-local variables are automatically deleted when the function ends.

:unl[et] ${env-name} ... :unlet-environment :unlet-$

Remove environment variable {env-name}.
Can mix {name} and ${env-name} in one :unlet command.
No error message is given for a non-existing variable, also without !.
If the system does not support deleting an environment variable, it is made empty.

:cons :const

```
:cons[t] {var-name} = {expr1}
:cons[t] [{name1}, {name2}, ...] = {expr1}
:cons[t] [{name}, ..., ; {lastname}] = {expr1}
:cons[t] {var-name} =<< [trim] [eval] {endmarker}
text...
text...
{endmarker}
```

Similar to :let, but additionally lock the variable after setting the value. This is the same as locking the variable with :lockvar just after :let, thus:

```
:const x = 1
```

is equivalent to:

```
:let x = 1
:lockvar! x
```

This is useful if you want to make sure the variable is not modified. If the value is a List or Dictionary literal then the items also cannot be changed:

```
const ll = [1, 2, 3]
let ll[1] = 5  " Error!
```

Nested references are not locked:

```
let lvar = ['a']
const lconst = [0, lvar]
let lconst[0] = 2  " Error!
let lconst[1][0] = 'b'  " OK
```

E995

It is an error to specify an existing variable with :const.

```
:let x = 1
:const x = 1  " Error!
```

E996

Note that environment variables, option values and register values cannot be used here, since they cannot be locked.

:cons[t]
:cons[t] {var-name}

If no argument is given or only {var-name} is given, the behavior is the same as :let.

:lockv[ar][!] [depth] {name} ... :lockvar :lockv

Lock the internal variable {name}. Locking means that it can no longer be changed (until it is unlocked). A locked variable can be deleted:

```
:lockvar v
:let v = 'asdf'   " fails!
:unlet v          " works
```

E741 E940 E1122

If you try to change a locked variable you get an error message: "E741: Value is locked: {name}". If you try to lock or unlock a built-in variable you will get an error message "E940: Cannot lock or unlock variable {name}".

[depth] is relevant when locking a List or Dictionary. It specifies how deep the locking goes:

0	Lock the variable {name} but not its value.
1	Lock the List or Dictionary itself, cannot add or remove items, but can still change their values.
2	Also lock the values, cannot change the items. If an item is a List or Dictionary, cannot add or remove items, but can still change the values.
3	Like 2 but for the List / Dictionary in the List / Dictionary, one level deeper.

The default [depth] is 2, thus when {name} is a List or Dictionary the values cannot be changed.

Example with [depth] 0:

```
let mylist = [1, 2, 3]
lockvar 0 mylist
let mylist[0] = 77      " OK
call add(mylist, 4)     " OK
let mylist = [7, 8, 9]  " Error!
```

E743

For unlimited depth use [!] and omit [depth]. However, there is a maximum depth of 100 to catch loops.

Note that when two variables refer to the same List and you lock one of them, the List will also be locked when used through the other variable. Example:

```
:let l = [0, 1, 2, 3]
:let cl = l
:lockvar l
:let cl[1] = 99         " won't work!
```

You may want to make a copy of a list to avoid this. See deepcopy().

:unlo[ckvar][!] [depth] {name} ... :unlockvar :unlo

Unlock the internal variable {name}. Does the opposite of :lockvar.

No error is given if {name} does not exist.

:if {expr1} :if :end :endif :en E171 E579 E580
:en[dif]

Execute the commands until the next matching :else or :endif if {expr1} evaluates to non-zero. Although the short forms work, it is recommended to always use :endif to avoid confusion and to make auto-indenting work properly.

From Vim version 4.5 until 5.0, every Ex command in between the :if and :endif is ignored. These two commands were just to allow for future expansions in a backward compatible way. Nesting was allowed. Note that any :else or :elseif was ignored, the else part was not executed either.

You can use this to remain compatible with older versions:

```
:if version >= 500
:  version-5-specific-commands
:endif
```

The commands still need to be parsed to find the endif. Sometimes an older Vim has a problem with a new command. For example, :silent is recognized as

a :substitute command. In that case :execute can
avoid problems:

```
:if version >= 600
:  execute "silent 1,$delete"
:endif
```

NOTE: The :append and :insert commands don't work
properly in between :if and :endif.

:else :el E581 E583

:el[se] Execute the commands until the next matching :else
or :endif if they previously were not being
executed.

:elseif :elsei E582 E584

:elsei[f] {expr1} Short for :else :if, with the addition that there
is no extra :endif.

:while :endwhile :wh :endw
E170 E585 E588 E733

:wh[ile] {expr1}

:endw[hile] Repeat the commands between :while and :endwhile,
as long as {expr1} evaluates to non-zero.
When an error is detected from a command inside the
loop, execution continues after the endwhile.
Example:

```
:let lnum = 1
:while lnum <= line("$")
   :call FixLine(lnum)
   :let lnum = lnum + 1
:endwhile
```

NOTE: The :append and :insert commands don't work
properly inside a :while and :for loop.

:for E690 E732

:for {var} in {object}

:endfo :endfor

:endfo[r] Repeat the commands between :for and :endfor for
each item in {object}. {object} can be a List,
a Blob or a String.

Variable {var} is set to the value of each item.

When an error is detected for a command inside the
loop, execution continues after the endfor.
Changing {object} inside the loop affects what items
are used. Make a copy if this is unwanted:

```
:for item in copy(mylist)
```

When {object} is a List and not making a copy, Vim
stores a reference to the next item in the List
before executing the commands with the current item.
Thus the current item can be removed without effect.
Removing any later item means it will not be found.
Thus the following example works (an inefficient way
to make a List empty):

```
for item in mylist
   call remove(mylist, 0)
endfor
```

Note that reordering the List (e.g., with sort() or
reverse()) may have unexpected effects.

When {object} is a Blob, Vim always makes a copy to
iterate over. Unlike with List, modifying the
Blob does not affect the iteration.

When {object} is a String each item is a string with
one character, plus any combining characters.

:for [{var1}, {var2}, ...] in {listlist}
:endfo[r]

Like :for above, but each item in {listlist} must be
a list, of which each item is assigned to {var1},
{var2}, etc. Example:

```
:for [lnum, col] in [[1, 3], [2, 5], [3, 8]]
   :echo getline(lnum)[col]
:endfor
```

:continue :con E586

:con[tinue] When used inside a :while or :for loop, jumps back
to the start of the loop.

If it is used after a :try inside the loop but
before the matching :finally (if present), the
commands following the :finally up to the matching
:endtry are executed first. This process applies to
all nested :trys inside the loop. The outermost
:endtry then jumps back to the start of the loop.

:break :brea E587

:brea[k] When used inside a :while or :for loop, skips to

the command after the matching :endwhile or
:endfor.
If it is used after a :try inside the loop but
before the matching :finally (if present), the
commands following the :finally up to the matching
:endtry are executed first. This process applies to
all nested :trys inside the loop. The outermost
:endtry then jumps to the command after the loop.

:try :endt :endtry E600 E601 E602

:endt[ry] Change the error handling for the commands between
:try and :endtry including everything being
executed across :source commands, function calls,
or autocommand invocations.

When an error or interrupt is detected and there is
a :finally command following, execution continues
after the :finally. Otherwise, or when the
:endtry is reached thereafter, the next
(dynamically) surrounding :try is checked for
a corresponding :finally etc. Then the script
processing is terminated. Whether a function
definition has an "abort" argument does not matter.
Example:

```
try | call Unknown() | finally | echomsg "cleanup" | endtry
echomsg "not reached"
```

Moreover, an error or interrupt (dynamically) inside
:try and :endtry is converted to an exception. It
can be caught as if it were thrown by a :throw
command (see :catch). In this case, the script
processing is not terminated.

The value "Vim:Interrupt" is used for an interrupt
exception. An error in a Vim command is converted
to a value of the form "Vim({command}):{errmsg}",
other errors are converted to a value of the form
"Vim:{errmsg}". {command} is the full command name,
and {errmsg} is the message that is displayed if the
error exception is not caught, always beginning with
the error number.
Examples:

```
try | sleep 100 | catch /^Vim:Interrupt$/ | endtry
try | edit | catch /^Vim(edit):E\d\+/ | echo "error" | endtry
```

:cat :catch
E603 E604 E605 E654

:cat[ch] [/{pattern}/] The following commands until the next :catch,
:finally, or :endtry that belongs to the same
:try as the :catch are executed when an exception
matching {pattern} is being thrown and has not yet
been caught by a previous :catch. Otherwise, these
commands are skipped.
Pattern can start with "Vim({cmd})" to indicate an
exception that occurred when executing the Ex command
{cmd}. When {pattern} is omitted all errors are
caught. Examples:

```
:catch /^Vim:Interrupt$/         " catch interrupts (CTRL-C)
:catch /^Vim\%((\S\+)\)\=:E/     " catch all Vim errors
:catch /^Vim\%((\S\+)\)\=:/      " catch errors and interrupts
:catch /^Vim(write):/            " catch all errors in :write
:catch /^Vim(!):/                " catch all errors in :!
:catch /^Vim\%((\S\+)\)\=:E123:/ " catch error E123
:catch /my-exception/            " catch user exception
:catch /.*/                      " catch everything
:catch                           " same as /.*/
```

Another character can be used instead of / around the
{pattern}, so long as it does not have a special
meaning (e.g., '|' or '"') and doesn't occur inside
{pattern}.
Information about the exception is available in
v:exception. Also see throw-variables.
NOTE: It is not reliable to ":catch" the TEXT of
an error message because it may vary in different
locales.

:fina :finally E606 E607

:fina[lly] The following commands until the matching :endtry
are executed whenever the part between the matching
:try and the :finally is left: either by falling
through to the :finally or by a :continue,
:break, :finish, or :return, or by an error or
interrupt or exception (see :throw).

:th :throw E608

:th[row] {expr1} The {expr1} is evaluated and thrown as an exception.
If the :throw is used after a :try but before the
first corresponding :catch, commands are skipped

until the first :catch matching {expr1} is reached.
If there is no such :catch or if the :throw is
used after a :catch but before the :finally, the
commands following the :finally (if present) up to
the matching :endtry are executed. If the :throw
is after the :finally, commands up to the :endtry
are skipped. At the :endtry, this process applies
again for the next dynamically surrounding :try
(which may be found in a calling function or sourcing
script), until a matching :catch has been found.
If the exception is not caught, the command processing
is terminated.
Example:

```
:try | throw "oops" | catch /^oo/ | echo "caught" | endtry
```

Note that "catch" may need to be on a separate line
for when an error causes the parsing to skip the whole
line and not see the "|" that separates the commands.

:ec :echo

:ec[ho] {expr1} ... Echoes each {expr1}, with a space in between. The
first {expr1} starts on a new line.
Also see :comment.
Use "\n" to start a new line. Use "\r" to move the
cursor to the first column.
Uses the highlighting set by the :echohl command.
Cannot be followed by a comment.
Example:

```
:echo "the value of 'shell' is" &shell
```

:echo-redraw

A later redraw may make the message disappear again.
And since Vim mostly postpones redrawing until it's
finished with a sequence of commands this happens
quite often. To avoid that a command from before the
:echo causes a redraw afterwards (redraws are often
postponed until you type something), force a redraw
with the :redraw command. Example:

```
:new | redraw | echo "there is a new window"
```

:echo-self-refer

When printing nested containers echo prints second
occurrence of the self-referencing container using
"[...@level]" (self-referencing List) or
"{...@level}" (self-referencing Dict):

```
:let l = []
:call add(l, l)
:let l2 = []
:call add(l2, [l2])
:echo l l2
```

echoes "[[...@0]] [[[...@0]]]". Echoing "[l]" will
echo "[[[...@1]]]" because l first occurs at second
level.

:echon

:echon {expr1} ...   Echoes each {expr1}, without anything added. Also see
:comment.
Uses the highlighting set by the :echohl command.
Cannot be followed by a comment.
Example:

```
:echon "the value of 'shell' is " &shell
```

Note the difference between using :echo, which is a
Vim command, and :!echo, which is an external shell
command:

```
:!echo %               --> filename
```

The arguments of ":!" are expanded, see :_%.

```
:!echo "%"             --> filename or "filename"
```

Like the previous example. Whether you see the double
quotes or not depends on your 'shell'.

```
:echo %                --> nothing
```

The '%' is an illegal character in an expression.

```
:echo "%"              --> %
```

This just echoes the '%' character.

```
:echo expand("%")       --> filename
```

This calls the expand() function to expand the '%'.

:echoh :echohl

:echoh[l] {name} Use the highlight group {name} for the following :echo, :echon and :echomsg commands. Also used for the input() prompt. Example:

```
:echohl WarningMsg | echo "Don't panic!" | echohl None
```

Don't forget to set the group back to "None", otherwise all following echo's will be highlighted.

:echom :echomsg

:echom[sg] {expr1} ... Echo the expression(s) as a true message, saving the message in the message-history. Spaces are placed between the arguments as with the :echo command. But unprintable characters are displayed, not interpreted. The parsing works slightly different from :echo, more like :execute. All the expressions are first evaluated and concatenated before echoing anything. If expressions does not evaluate to a Number or String, string() is used to turn it into a string. Uses the highlighting set by the :echohl command. Example:

```
:echomsg "It's a Zizzer Zazzer Zuzz, as you can plainly see."
```

See :echo-redraw to avoid the message disappearing when the screen is redrawn.

:echoe :echoerr

:echoe[rr] {expr1} ... Echo the expression(s) as an error message, saving the message in the message-history. When used in a script or function the line number will be added. Spaces are placed between the arguments as with the :echomsg command. When used inside a try conditional, the message is raised as an error exception instead (see try-echoerr). Example:

```
:echoerr "This script just failed!"
```

If you just want a highlighted message use :echohl. And to get a beep:

```
:exe "normal \<Esc>"
```

:eval

:eval {expr} Evaluate {expr} and discard the result. Example:

```
:eval Getlist()->Filter()->append('$')
```

The expression is supposed to have a side effect, since the resulting value is not used. In the example the append() call appends the List with text to the buffer. This is similar to :call but works with any expression.

The command can be shortened to :ev or :eva, but these are hard to recognize and therefore not to be used.

The command cannot be followed by "|" and another command, since "|" is seen as part of the expression.

:exe :execute

:exe[cute] {expr1} ... Executes the string that results from the evaluation of {expr1} as an Ex command. Multiple arguments are concatenated, with a space in between. To avoid the extra space use the ".." operator to concatenate strings into one argument. {expr1} is used as the processed command, command line editing keys are not recognized. Cannot be followed by a comment. Examples:

```
:execute "buffer" nextbuf
:execute "normal" count .. "w"
```

":execute" can be used to append a command to commands that don't accept a '|'. Example:

```
:execute '!ls' | echo "theend"
```

":execute" is also a nice way to avoid having to type control characters in a Vim script for a ":normal" command:

```
:execute "normal ixxx\<Esc>"
```

This has an <Esc> character, see expr-string.

Be careful to correctly escape special characters in file names. The fnameescape() function can be used for Vim commands, shellescape() for :! commands. Examples:

```
:execute "e " .. fnameescape(filename)
:execute "!ls " .. shellescape(filename, 1)
```

Note: The executed string may be any command-line, but starting or ending "if", "while" and "for" does not always work, because when commands are skipped the ":execute" is not evaluated and Vim loses track of where blocks start and end. Also "break" and "continue" should not be inside ":execute". This example does not work, because the ":execute" is not evaluated and Vim does not see the "while", and gives an error for finding an ":endwhile":

```
:if 0
: execute 'while i > 5'
:  echo "test"
: endwhile
:endif
```

It is allowed to have a "while" or "if" command completely in the executed string:

```
:execute 'while i < 5 | echo i | let i = i + 1 | endwhile'
```

:exe-comment

":execute", ":echo" and ":echon" cannot be followed by a comment directly, because they see the '"' as the start of a string. But, you can use '|' followed by a comment. Example:

```
:echo "foo" | "this is a comment
```

8. Exception handling exception-handling

The Vim script language comprises an exception handling feature. This section explains how it can be used in a Vim script.

Exceptions may be raised by Vim on an error or on interrupt, see catch-errors and catch-interrupt. You can also explicitly throw an exception by using the ":throw" command, see throw-catch.

TRY CONDITIONALS try-conditionals

Exceptions can be caught or can cause cleanup code to be executed. You can use a try conditional to specify catch clauses (that catch exceptions) and/or a finally clause (to be executed for cleanup).
   A try conditional begins with a :try command and ends at the matching :endtry command. In between, you can use a :catch command to start a catch clause, or a :finally command to start a finally clause. There may be none or multiple catch clauses, but there is at most one finally clause, which must not be followed by any catch clauses. The lines before the catch clauses and the finally clause is called a try block.

```
:try
:       ...
:       ...                             TRY BLOCK
:       ...
:catch /{pattern}/
:       ...
:       ...                             CATCH CLAUSE
:       ...
:catch /{pattern}/
:       ...
:       ...                             CATCH CLAUSE
:       ...
:finally
:       ...
:       ...                             FINALLY CLAUSE
:       ...
:endtry
```

The try conditional allows to watch code for exceptions and to take the appropriate actions. Exceptions from the try block may be caught. Exceptions from the try block and also the catch clauses may cause cleanup actions.
   When no exception is thrown during execution of the try block, the control is transferred to the finally clause, if present. After its execution, the script continues with the line following the ":endtry".

When an exception occurs during execution of the try block, the remaining lines in the try block are skipped. The exception is matched against the patterns specified as arguments to the ":catch" commands. The catch clause after the first matching ":catch" is taken, other catch clauses are not executed. The catch clause ends when the next ":catch", ":finally", or ":endtry" command is reached - whatever is first. Then, the finally clause (if present) is executed. When the ":endtry" is reached, the script execution continues in the following line as usual.

When an exception that does not match any of the patterns specified by the ":catch" commands is thrown in the try block, the exception is not caught by that try conditional and none of the catch clauses is executed. Only the finally clause, if present, is taken. The exception pends during execution of the finally clause. It is resumed at the ":endtry", so that commands after the ":endtry" are not executed and the exception might be caught elsewhere, see try-nesting.

When during execution of a catch clause another exception is thrown, the remaining lines in that catch clause are not executed. The new exception is not matched against the patterns in any of the ":catch" commands of the same try conditional and none of its catch clauses is taken. If there is, however, a finally clause, it is executed, and the exception pends during its execution. The commands following the ":endtry" are not executed. The new exception might, however, be caught elsewhere, see try-nesting.

When during execution of the finally clause (if present) an exception is thrown, the remaining lines in the finally clause are skipped. If the finally clause has been taken because of an exception from the try block or one of the catch clauses, the original (pending) exception is discarded. The commands following the ":endtry" are not executed, and the exception from the finally clause is propagated and can be caught elsewhere, see try-nesting.

The finally clause is also executed, when a ":break" or ":continue" for a ":while" loop enclosing the complete try conditional is executed from the try block or a catch clause. Or when a ":return" or ":finish" is executed from the try block or a catch clause of a try conditional in a function or sourced script, respectively. The ":break", ":continue", ":return", or ":finish" pends during execution of the finally clause and is resumed when the ":endtry" is reached. It is, however, discarded when an exception is thrown from the finally clause.

When a ":break" or ":continue" for a ":while" loop enclosing the complete try conditional or when a ":return" or ":finish" is encountered in the finally clause, the rest of the finally clause is skipped, and the ":break", ":continue", ":return" or ":finish" is executed as usual. If the finally clause has been taken because of an exception or an earlier ":break", ":continue", ":return", or ":finish" from the try block or a catch clause, this pending exception or command is discarded.

For examples see throw-catch and try-finally.

NESTING OF TRY CONDITIONALS try-nesting

Try conditionals can be nested arbitrarily. That is, a complete try conditional can be put into the try block, a catch clause, or the finally clause of another try conditional. If the inner try conditional does not catch an exception thrown in its try block or throws a new exception from one of its catch clauses or its finally clause, the outer try conditional is checked according to the rules above. If the inner try conditional is in the try block of the outer try conditional, its catch clauses are checked, but otherwise only the finally clause is executed. It does not matter for nesting, whether the inner try conditional is directly contained in the outer one, or whether the outer one sources a script or calls a function containing the inner try conditional.

When none of the active try conditionals catches an exception, just their finally clauses are executed. Thereafter, the script processing terminates. An error message is displayed in case of an uncaught exception explicitly thrown by a ":throw" command. For uncaught error and interrupt exceptions implicitly raised by Vim, the error message(s) or interrupt message are shown as usual.

For examples see throw-catch.

EXAMINING EXCEPTION HANDLING CODE except-examine

Exception handling code can get tricky. If you are in doubt what happens, set 'verbose' to 13 or use the ":13verbose" command modifier when sourcing your script file. Then you see when an exception is thrown, discarded, caught, or finished. When using a verbosity level of at least 14, things pending in a finally clause are also shown. This information is also given in debug mode (see debug-scripts).

THROWING AND CATCHING EXCEPTIONS throw-catch

You can throw any number or string as an exception. Use the :throw command and pass the value to be thrown as argument:

```
:throw 4711
:throw "string"
```

throw-expression

You can also specify an expression argument. The expression is then evaluated first, and the result is thrown:

```
:throw 4705 + strlen("string")
:throw strpart("strings", 0, 6)
```

An exception might be thrown during evaluation of the argument of the ":throw"
command. Unless it is caught there, the expression evaluation is abandoned.
The ":throw" command then does not throw a new exception.
   Example:

```
:function! Foo(arg)
:  try
:    throw a:arg
:  catch /foo/
:  endtry
:  return 1
:endfunction
:
:function! Bar()
:  echo "in Bar"
:  return 4710
:endfunction
:
:throw Foo("arrgh") + Bar()
```

This throws "arrgh", and "in Bar" is not displayed since Bar() is not
executed.

```
:throw Foo("foo") + Bar()
```

however displays "in Bar" and throws 4711.

Any other command that takes an expression as argument might also be
abandoned by an (uncaught) exception during the expression evaluation. The
exception is then propagated to the caller of the command.
   Example:

```
:if Foo("arrgh")
:  echo "then"
:else
:  echo "else"
:endif
```

Here neither of "then" or "else" is displayed.

catch-order

Exceptions can be caught by a try conditional with one or more :catch
commands, see try-conditionals.   The values to be caught by each ":catch"
command can be specified as a pattern argument. The subsequent catch clause
gets executed when a matching exception is caught.
   Example:

```
:function! Foo(value)
:  try
:    throw a:value
:  catch /^\d\+$/
:    echo "Number thrown"
:  catch /.*/
:    echo "String thrown"
:  endtry
:endfunction
:
:call Foo(0x1267)
:call Foo('string')
```

The first call to Foo() displays "Number thrown", the second "String thrown".
An exception is matched against the ":catch" commands in the order they are
specified. Only the first match counts. So you should place the more
specific ":catch" first. The following order does not make sense:

```
:  catch /.*/
:    echo "String thrown"
:  catch /^\d\+$/
:    echo "Number thrown"
```

The first ":catch" here matches always, so that the second catch clause is
never taken.

throw-variables

If you catch an exception by a general pattern, you may access the exact value
in the variable v:exception:

```
:  catch /^\d\+$/
:    echo "Number thrown.  Value is" v:exception
```

You may also be interested where an exception was thrown. This is stored in
v:throwpoint. And you can obtain the stack trace from v:stacktrace.
Note that "v:exception", "v:stacktrace" and "v:throwpoint" are valid for the
exception most recently caught as long it is not finished.
   Example:

```
:function! Caught()
:  if v:exception != ""
:    echo 'Caught "' .. v:exception .. '" in ' .. v:throwpoint
:  else
:    echo 'Nothing caught'
:  endif
:endfunction
:
:function! Foo()
:  try
:    try
:      try
:        throw 4711
:      finally
:        call Caught()
:      endtry
:    catch /.*/
:      call Caught()
:      throw "oops"
:    endtry
:  catch /.*/
:    call Caught()
:  finally
:    call Caught()
:  endtry
:endfunction
:
:call Foo()
```

This displays

```
Nothing caught
Caught "4711" in function Foo, line 4
Caught "oops" in function Foo, line 10
Nothing caught
```

A practical example: The following command ":LineNumber" displays the line
number in the script or function where it has been used:

```
:function! LineNumber()
:    return substitute(v:throwpoint, '.*\D\(\d\+\).*', '\1', "")
:endfunction
:command! LineNumber try | throw "" | catch | echo LineNumber() | endtry
```

try-nested

An exception that is not caught by a try conditional can be caught by
a surrounding try conditional:

```
:try
:  try
:    throw "foo"
:  catch /foobar/
:    echo "foobar"
:  finally
:    echo "inner finally"
:  endtry
:catch /foo/
:  echo "foo"
:endtry
```

The inner try conditional does not catch the exception, just its finally
clause is executed. The exception is then caught by the outer try
conditional. The example displays "inner finally" and then "foo".

throw-from-catch

You can catch an exception and throw a new one to be caught elsewhere from the
catch clause:

```
:function! Foo()
:  throw "foo"
:endfunction
:
:function! Bar()
:  try
:    call Foo()
:  catch /foo/
:    echo "Caught foo, throw bar"
:    throw "bar"
:  endtry
:endfunction
:
:try
:  call Bar()
:catch /.*/
:  echo "Caught" v:exception
:endtry
```

This displays "Caught foo, throw bar" and then "Caught bar".

rethrow

There is no real rethrow in the Vim script language, but you may throw
"v:exception" instead:

```
:function! Bar()
:  try
:    call Foo()
:  catch /.*/
:    echo "Rethrow" v:exception
:    throw v:exception
:  endtry
:endfunction
```

try-echoerr

Note that this method cannot be used to "rethrow" Vim error or interrupt
exceptions, because it is not possible to fake Vim internal exceptions.
Trying so causes an error exception. You should throw your own exception
denoting the situation. If you want to cause a Vim error exception containing
the original error exception value, you can use the :echoerr command:

```
:try
:  try
:    asdf
:  catch /.*/
:    echoerr v:exception
:  endtry
:catch /.*/
:  echo v:exception
:endtry
```

This code displays

Vim(echoerr):Vim:E492: Not an editor command: asdf

CLEANUP CODE **try-finally**

Scripts often change global settings and restore them at their end. If the
user however interrupts the script by pressing CTRL-C, the settings remain in
an inconsistent state. The same may happen to you in the development phase of
a script when an error occurs or you explicitly throw an exception without
catching it. You can solve these problems by using a try conditional with
a finally clause for restoring the settings. Its execution is guaranteed on
normal control flow, on error, on an explicit ":throw", and on interrupt.
(Note that errors and interrupts from inside the try conditional are converted
to exceptions. When not caught, they terminate the script after the finally
clause has been executed.)
Example:

```
:try
:  let s:saved_ts = &ts
:  set ts=17
:
:  " Do the hard work here.
:
:finally
:  let &ts = s:saved_ts
:  unlet s:saved_ts
:endtry
```

This method should be used locally whenever a function or part of a script
changes global settings which need to be restored on failure or normal exit of
that function or script part.

break-finally

Cleanup code works also when the try block or a catch clause is left by
a ":continue", ":break", ":return", or ":finish".
   Example:

```
:let first = 1
:while 1
:  try
:    if first
:      echo "first"
:      let first = 0
:      continue
:    else
:      throw "second"
:    endif
:  catch /.*/
:    echo v:exception
:    break
:  finally
:    echo "cleanup"
:  endtry
:  echo "still in while"
:endwhile
:echo "end"
```

This displays "first", "cleanup", "second", "cleanup", and "end".

```
:function! Foo()
:  try
:    return 4711
:  finally
:    echo "cleanup\n"
:  endtry
:  echo "Foo still active"
:endfunction
:
:echo Foo() "returned by Foo"
```

This displays "cleanup" and "4711 returned by Foo". You don't need to add an
extra ":return" in the finally clause. (Above all, this would override the
return value.)

except-from-finally

Using either of ":continue", ":break", ":return", ":finish", or ":throw" in
a finally clause is possible, but not recommended since it abandons the
cleanup actions for the try conditional. But, of course, interrupt and error
exceptions might get raised from a finally clause.
   Example where an error in the finally clause stops an interrupt from
working correctly:

```
:try
:  try
:    echo "Press CTRL-C for interrupt"
:    while 1
:    endwhile
:  finally
:    unlet novar
:  endtry
:catch /novar/
:endtry
:echo "Script still running"
:sleep 1
```

If you need to put commands that could fail into a finally clause, you should
think about catching or ignoring the errors in these commands, see
catch-errors and ignore-errors.

CATCHING ERRORS **catch-errors**

If you want to catch specific errors, you just have to put the code to be
watched in a try block and add a catch clause for the error message. The
presence of the try conditional causes all errors to be converted to an
exception. No message is displayed and v:errmsg is not set then. To find
the right pattern for the ":catch" command, you have to know how the format of
the error exception is.
   Error exceptions have the following format:

```
Vim({cmdname}):{errmsg}
```

or

```
Vim:{errmsg}
```

{cmdname} is the name of the command that failed; the second form is used when
the command name is not known. {errmsg} is the error message usually produced

when the error occurs outside try conditionals. It always begins with
a capital "E", followed by a two or three-digit error number, a colon, and
a space.

Examples:

The command

```
:unlet novar
```

normally produces the error message

```
E108: No such variable: "novar"
```

which is converted inside try conditionals to an exception

```
Vim(unlet):E108: No such variable: "novar"
```

The command

```
:dwim
```

normally produces the error message

```
E492: Not an editor command: dwim
```

which is converted inside try conditionals to an exception

```
Vim:E492: Not an editor command: dwim
```

You can catch all ":unlet" errors by a

```
:catch /^Vim(unlet):/
```

or all errors for misspelled command names by a

```
:catch /^Vim:E492:/
```

Some error messages may be produced by different commands:

```
:function nofunc
```

and

```
:delfunction nofunc
```

both produce the error message

```
E128: Function name must start with a capital: nofunc
```

which is converted inside try conditionals to an exception

```
Vim(function):E128: Function name must start with a capital: nofunc
```

or

```
Vim(delfunction):E128: Function name must start with a capital: nofunc
```

respectively. You can catch the error by its number independently on the
command that caused it if you use the following pattern:

```
:catch /^Vim(\a\+):E128:/
```

Some commands like

```
:let x = novar
```

produce multiple error messages, here:

```
E121: Undefined variable: novar
E15: Invalid expression:  novar
```

Only the first is used for the exception value, since it is the most specific
one (see except-several-errors). So you can catch it by

```
:catch /^Vim(\a\+):E121:/
```

You can catch all errors related to the name "nofunc" by

```
:catch /\<nofunc\>/
```

You can catch all Vim errors in the ":write" and ":read" commands by

```
:catch /^Vim(\(write\|read\)):E\d\+:/
```

You can catch all Vim errors by the pattern

```
:catch /^Vim\((\a\+)\)\=:E\d\+:/
```

catch-text

NOTE: You should never catch the error message text itself:

```
:catch /No such variable/
```

only works in the English locale, but not when the user has selected
a different language by the :language command. It is however helpful to
cite the message text in a comment:

```
:catch /^Vim(\a\+):E108:/   " No such variable
```

IGNORING ERRORS

ignore-errors

You can ignore errors in a specific Vim command by catching them locally:

```
:try
:  write
:catch
:endtry
```

But you are strongly recommended NOT to use this simple form, since it could
catch more than you want. With the ":write" command, some autocommands could
be executed and cause errors not related to writing, for instance:

```
:au BufWritePre * unlet novar
```

There could even be such errors you are not responsible for as a script
writer: a user of your script might have defined such autocommands. You would
then hide the error from the user.
   It is much better to use

```
:try
:  write
:catch /^Vim(write):/
:endtry
```

which only catches real write errors. So catch only what you'd like to ignore
intentionally.

For a single command that does not cause execution of autocommands, you could
even suppress the conversion of errors to exceptions by the ":silent!"
command:

```
:silent! nunmap k
```

This works also when a try conditional is active.

CATCHING INTERRUPTS

catch-interrupt

When there are active try conditionals, an interrupt (CTRL-C) is converted to
the exception "Vim:Interrupt". You can catch it like every exception. The
script is not terminated, then.
   Example:

```
:function! TASK1()
:  sleep 10
:endfunction

:function! TASK2()
:  sleep 20
:endfunction

:while 1
:  let command = input("Type a command: ")
:  try
:    if command == ""
:      continue
:    elseif command == "END"
:      break
:    elseif command == "TASK1"
:      call TASK1()
:    elseif command == "TASK2"
:      call TASK2()
:    else
:      echo "\nIllegal command:" command
:      continue
:    endif
:  catch /^Vim:Interrupt$/
:    echo "\nCommand interrupted"
:    " Caught the interrupt.  Continue with next prompt.
:  endtry
:endwhile
```

You can interrupt a task here by pressing CTRL-C; the script then asks for
a new command. If you press CTRL-C at the prompt, the script is terminated.

For testing what happens when CTRL-C would be pressed on a specific line in
your script, use the debug mode and execute the >quit or >interrupt
command on that line. See debug-scripts.

CATCHING ALL **catch-all**

The commands

```
:catch /.*/
:catch //
:catch
```

catch everything, error exceptions, interrupt exceptions and exceptions
explicitly thrown by the :throw command. This is useful at the top level of
a script in order to catch unexpected things.
   Example:

```
:try
:
:  " do the hard work here
:
:catch /MyException/
:
:  " handle known problem
:
:catch /^Vim:Interrupt$/
:    echo "Script interrupted"
:catch /.*/
:  echo "Internal error (" .. v:exception .. ")"
:  echo " - occurred at " .. v:throwpoint
:endtry
:" end of script
```

Note: Catching all might catch more things than you want. Thus, you are
strongly encouraged to catch only for problems that you can really handle by
specifying a pattern argument to the ":catch".
   Example: Catching all could make it nearly impossible to interrupt a script
by pressing CTRL-C:

```
:while 1
:  try
:    sleep 1
:  catch
:  endtry
:endwhile
```

EXCEPTIONS AND AUTOCOMMANDS **except-autocmd**

Exceptions may be used during execution of autocommands. Example:

```
:autocmd User x try
:autocmd User x   throw "Oops!"
:autocmd User x catch
:autocmd User x   echo v:exception
:autocmd User x endtry
:autocmd User x throw "Arrgh!"
:autocmd User x echo "Should not be displayed"
:
:try
:  doautocmd User x
:catch
:  echo v:exception
:endtry
```

This displays "Oops!" and "Arrgh!".

except-autocmd-Pre

For some commands, autocommands get executed before the main action of the command takes place. If an exception is thrown and not caught in the sequence of autocommands, the sequence and the command that caused its execution are abandoned and the exception is propagated to the caller of the command.
Example:

```
:autocmd BufWritePre * throw "FAIL"
:autocmd BufWritePre * echo "Should not be displayed"
:
:try
:  write
:catch
:  echo "Caught:" v:exception "from" v:throwpoint
:endtry
```

Here, the ":write" command does not write the file currently being edited (as you can see by checking 'modified'), since the exception from the BufWritePre autocommand abandons the ":write". The exception is then caught and the script displays:

```
Caught: FAIL from BufWrite Auto commands for "*"
```

except-autocmd-Post

For some commands, autocommands get executed after the main action of the command has taken place. If this main action fails and the command is inside an active try conditional, the autocommands are skipped and an error exception is thrown that can be caught by the caller of the command.
Example:

```
:autocmd BufWritePost * echo "File successfully written!"
:
:try
:  write /i/m/p/o/s/s/i/b/l/e
:catch
:  echo v:exception
:endtry
```

This just displays:

```
Vim(write):E212: Can't open file for writing (/i/m/p/o/s/s/i/b/l/e)
```

If you really need to execute the autocommands even when the main action fails, trigger the event from the catch clause.
Example:

```
:autocmd BufWritePre  * set noreadonly
:autocmd BufWritePost * set readonly
:
:try
:  write /i/m/p/o/s/s/i/b/l/e
:catch
:  doautocmd BufWritePost /i/m/p/o/s/s/i/b/l/e
:endtry
```

You can also use ":silent!":

```
:let x = "ok"
:let v:errmsg = ""
:autocmd BufWritePost * if v:errmsg != ""
:autocmd BufWritePost *   let x = "after fail"
:autocmd BufWritePost * endif
:try
:  silent! write /i/m/p/o/s/s/i/b/l/e
:catch
:endtry
:echo x
```

This displays "after fail".

If the main action of the command does not fail, exceptions from the
autocommands will be catchable by the caller of the command:

```
:autocmd BufWritePost * throw ":-("
:autocmd BufWritePost * echo "Should not be displayed"
:
:try
:  write
:catch
:  echo v:exception
:endtry
```

except-autocmd-Cmd

For some commands, the normal action can be replaced by a sequence of
autocommands. Exceptions from that sequence will be catchable by the caller
of the command.
   Example: For the ":write" command, the caller cannot know whether the file
had actually been written when the exception occurred. You need to tell it in
some way.

```
:if !exists("cnt")
:  let cnt = 0
:
:  autocmd BufWriteCmd * if &modified
:  autocmd BufWriteCmd *   let cnt = cnt + 1
:  autocmd BufWriteCmd *   if cnt % 3 == 2
:  autocmd BufWriteCmd *     throw "BufWriteCmdError"
:  autocmd BufWriteCmd *   endif
:  autocmd BufWriteCmd *   write | set nomodified
:  autocmd BufWriteCmd *   if cnt % 3 == 0
:  autocmd BufWriteCmd *     throw "BufWriteCmdError"
:  autocmd BufWriteCmd *   endif
:  autocmd BufWriteCmd *   echo "File successfully written!"
:  autocmd BufWriteCmd * endif
:endif
:
:try
:       write
:catch /^BufWriteCmdError$/
:  if &modified
:    echo "Error on writing (file contents not changed)"
:  else
:    echo "Error after writing"
:  endif
:catch /^Vim(write):/
:    echo "Error on writing"
:endtry
```

When this script is sourced several times after making changes, it displays
first

```
File successfully written!
```

then

```
Error on writing (file contents not changed)
```

then

```
Error after writing
```

etc.

except-autocmd-ill

You cannot spread a try conditional over autocommands for different events.
The following code is ill-formed:

```
:autocmd BufWritePre  * try
:
:autocmd BufWritePost * catch
:autocmd BufWritePost *   echo v:exception
:autocmd BufWritePost * endtry
:
:write
```

EXCEPTION HIERARCHIES AND PARAMETERIZED EXCEPTIONS **except-hier-param**

Some programming languages allow to use hierarchies of exception classes or to
pass additional information with the object of an exception class. You can do
similar things in Vim.

In order to throw an exception from a hierarchy, just throw the complete class name with the components separated by a colon, for instance throw the string "EXCEPT:MATHERR:OVERFLOW" for an overflow in a mathematical library.
When you want to pass additional information with your exception class, add it in parentheses, for instance throw the string "EXCEPT:IO:WRITEERR(myfile)" for an error when writing "myfile".
With the appropriate patterns in the ":catch" command, you can catch for base classes or derived classes of your hierarchy. Additional information in parentheses can be cut out from v:exception with the ":substitute" command.
Example:

```
:function! CheckRange(a, func)
:  if a:a < 0
:    throw "EXCEPT:MATHERR:RANGE(" .. a:func .. ")"
:  endif
:endfunction
:
:function! Add(a, b)
:  call CheckRange(a:a, "Add")
:  call CheckRange(a:b, "Add")
:  let c = a:a + a:b
:  if c < 0
:    throw "EXCEPT:MATHERR:OVERFLOW"
:  endif
:  return c
:endfunction
:
:function! Div(a, b)
:  call CheckRange(a:a, "Div")
:  call CheckRange(a:b, "Div")
:  if (a:b == 0)
:    throw "EXCEPT:MATHERR:ZERODIV"
:  endif
:  return a:a / a:b
:endfunction
:
:function! Write(file)
:  try
:    execute "write" fnameescape(a:file)
:  catch /^Vim(write):/
:    throw "EXCEPT:IO(" .. getcwd() .. ", " .. a:file .. "):WRITEERR"
:  endtry
:endfunction
:
:try
:
:  " something with arithmetic and I/O
:
:catch /^EXCEPT:MATHERR:RANGE/
:  let function = substitute(v:exception, '.*(\(\a\+\)).*', '\1', "")
:  echo "Range error in" function
:
:catch /^EXCEPT:MATHERR/        " catches OVERFLOW and ZERODIV
:  echo "Math error"
:
:catch /^EXCEPT:IO/
:  let dir = substitute(v:exception, '.*(\(.\+\),\s*.\+).*', '\1', "")
:  let file = substitute(v:exception, '.*(.\+,\s*\(.\+\)).*', '\1', "")
:  if file !~ '^/'
:    let file = dir .. "/" .. file
:  endif
:  echo 'I/O error for "' .. file .. '"'
:
:catch /^EXCEPT/
:  echo "Unspecified error"
:
:endtry
```

The exceptions raised by Vim itself (on error or when pressing CTRL-C) use a flat hierarchy: they are all in the "Vim" class. You cannot throw yourself exceptions with the "Vim" prefix; they are reserved for Vim.
Vim error exceptions are parameterized with the name of the command that failed, if known. See catch-errors.

PECULIARITIES

except-compat

The exception handling concept requires that the command sequence causing the exception is aborted immediately and control is transferred to finally clauses and/or a catch clause.

In the Vim script language there are cases where scripts and functions continue after an error: in functions without the "abort" flag or in a command after ":silent!", control flow goes to the following line, and outside functions, control flow goes to the line following the outermost ":endwhile" or ":endif". On the other hand, errors should be catchable as exceptions (thus, requiring the immediate abortion).

This problem has been solved by converting errors to exceptions and using
immediate abortion (if not suppressed by ":silent!") only when a try
conditional is active. This is no restriction since an (error) exception can
be caught only from an active try conditional. If you want an immediate
termination without catching the error, just use a try conditional without
catch clause. (You can cause cleanup code being executed before termination
by specifying a finally clause.)

When no try conditional is active, the usual abortion and continuation
behavior is used instead of immediate abortion. This ensures compatibility of
scripts written for Vim 6.1 and earlier.

However, when sourcing an existing script that does not use exception handling
commands (or when calling one of its functions) from inside an active try
conditional of a new script, you might change the control flow of the existing
script on error. You get the immediate abortion on error and can catch the
error in the new script. If however the sourced script suppresses error
messages by using the ":silent!" command (checking for errors by testing
v:errmsg if appropriate), its execution path is not changed. The error is
not converted to an exception. (See :silent.) So the only remaining cause
where this happens is for scripts that don't care about errors and produce
error messages. You probably won't want to use such code from your new
scripts.

except-syntax-err

Syntax errors in the exception handling commands are never caught by any of
the ":catch" commands of the try conditional they belong to. Its finally
clauses, however, is executed.
   Example:

```
:try
:  try
:    throw 4711
:  catch /\(/
:    echo "in catch with syntax error"
:  catch
:    echo "inner catch-all"
:  finally
:    echo "inner finally"
:  endtry
:catch
:  echo 'outer catch-all caught "' .. v:exception .. '"'
:  finally
:    echo "outer finally"
:endtry
```

This displays:

```
inner finally
outer catch-all caught "Vim(catch):E54: Unmatched \("
outer finally
```

The original exception is discarded and an error exception is raised, instead.

except-single-line

The ":try", ":catch", ":finally", and ":endtry" commands can be put on
a single line, but then syntax errors may make it difficult to recognize the
"catch" line, thus you better avoid this.
   Example:

```
:try | unlet! foo # | catch | endtry
```

raises an error exception for the trailing characters after the ":unlet!"
argument, but does not see the ":catch" and ":endtry" commands, so that the
error exception is discarded and the "E488: Trailing characters" message gets
displayed.

except-several-errors

When several errors appear in a single command, the first error message is
usually the most specific one and therefore converted to the error exception.
   Example:

```
echo novar
```

causes

```
E121: Undefined variable: novar
E15: Invalid expression: novar
```

The value of the error exception inside try conditionals is:

```
Vim(echo):E121: Undefined variable: novar
```

except-syntax-error

But when a syntax error is detected after a normal error in the same command,
the syntax error is used for the exception being thrown.
   Example:

```
unlet novar #
```

causes

```
E108: No such variable: "novar"
E488: Trailing characters
```

The value of the error exception inside try conditionals is:

```
Vim(unlet):E488: Trailing characters
```

This is done because the syntax error might change the execution path in a way
not intended by the user. Example:

```
try
    try | unlet novar # | catch | echo v:exception | endtry
catch /.*/
    echo "outer catch:" v:exception
endtry
```

This displays "outer catch: Vim(unlet):E488: Trailing characters", and then
a "E600: Missing :endtry" error message is given, see except-single-line.

9. Examples eval-examples

Printing in Binary

```
:" The function Nr2Bin() returns the binary string representation of a number.
:func Nr2Bin(nr)
:  let n = a:nr
:  let r = ""
:  while n
:    let r = '01'[n % 2] .. r
:    let n = n / 2
:  endwhile
:  return r
:endfunc

:" The function String2Bin() converts each character in a string to a
:" binary string, separated with dashes.
:func String2Bin(str)
:  let out = ''
:  for ix in range(strlen(a:str))
:    let out = out .. '-' .. Nr2Bin(char2nr(a:str[ix]))
:  endfor
:  return out[1:]
:endfunc
```

Example of its use:

```
:echo Nr2Bin(32)
```

result: "100000"

```
:echo String2Bin("32")
```

result: "110011-110010"

Sorting lines

This example sorts lines with a specific compare function.

```
:func SortBuffer()
:  let lines = getline(1, '$')
:  call sort(lines, function("Strcmp"))
:  call setline(1, lines)
:endfunction
```

As a one-liner:

```
:call setline(1, sort(getline(1, '$'), function("Strcmp")))
```

scanf() replacement

sscanf

There is no sscanf() function in Vim. If you need to extract parts from a
line, you can use matchstr() and substitute() to do it. This example shows

how to get the file name, line number and column number out of a line like
"foobar.txt, 123, 45".

```
:" Set up the match bit
:let mx='\(\f\+\),\s*\(\d\+\),\s*\(\d\+\)'
:"get the part matching the whole expression
:let l = matchstr(line, mx)
:"get each item out of the match
:let file = substitute(l, mx, '\1', '')
:let lnum = substitute(l, mx, '\2', '')
:let col = substitute(l, mx, '\3', '')
```

The input is in the variable "line", the results in the variables "file",
"lnum" and "col". (idea from Michael Geddes)

getting the scriptnames in a Dictionary

scriptnames-dictionary

The :scriptnames command can be used to get a list of all script files that
have been sourced. There is also the getscriptinfo() function, but the
information returned is not exactly the same. In case you need to manipulate
the output of scriptnames this code can be used:

```
" Get the output of ":scriptnames" in the scriptnames_output variable.
let scriptnames_output = ''
redir => scriptnames_output
silent scriptnames
redir END

" Split the output into lines and parse each line.      Add an entry to the
" "scripts" dictionary.
let scripts = {}
for line in split(scriptnames_output, "\n")
  " Only do non-blank lines.
  if line =~ '\S'
    " Get the first number in the line.
    let nr = matchstr(line, '\d\+')
    " Get the file name, remove the script number " 123: ".
    let name = substitute(line, '.\+:\s*', '', '')
    " Add an item to the Dictionary
    let scripts[nr] = name
  endif
endfor
unlet scriptnames_output
```

The sandbox

eval-sandbox sandbox E523

The 'foldexpr', 'formatexpr', 'includeexpr', 'indentexpr', 'statusline' and
'foldtext' options may be evaluated in a sandbox. This means that you are
protected from these expressions having nasty side effects. This gives some
safety for when these options are set from a modeline. It is also used when
the command from a tags file is executed and for CTRL-R = in the command line.
The sandbox is also used for the :sandbox command.

E48

These items are not allowed in the sandbox:

- changing the buffer text
- defining or changing mapping, autocommands, user commands
- setting certain options (see option-summary)
- setting certain v: variables (see v:var) E794
- executing a shell command
- reading or writing a file
- jumping to another buffer or editing a file
- executing Python, Perl, etc. commands

  This is not guaranteed 100% secure, but it should block most attacks.

:san :sandbox

:san[dbox] {cmd} Execute {cmd} in the sandbox. Useful to evaluate an option that may have been set from a modeline, e.g. 'foldexpr'.

sandbox-option

A few options contain an expression. When this expression is evaluated it may have to be done in the sandbox to avoid a security risk. But the sandbox is restrictive, thus this only happens when the option was set from an insecure location. Insecure in this context are:

- sourcing a .nvimrc or .exrc in the current directory
- while executing in the sandbox
- value coming from a modeline
- executing a function that was defined in the sandbox

Note that when in the sandbox and saving an option value and restoring it, the option will still be marked as it was set in the sandbox.

Textlock

textlock

In a few situations it is not allowed to change the text in the buffer, jump to another window and some other things that might confuse or break what Vim is currently doing. This mostly applies to things that happen when Vim is actually doing something else. For example, a TextYankPost autocommand cannot edit the text it is yanking.

This is not allowed when the textlock is active:

- changing the buffer text
- jumping to another buffer or window
- editing another file
- closing a window or quitting Vim
- etc.

Vim script library

vimscript-library

Vim comes bundled with a Vim script library, that can be used by runtime, script authors. Currently, it only includes very few functions, but it may grow over time.

dist#vim

The functions make use of the autoloaded prefix "dist#vim".

The following functions are available:

dist#vim#IsSafeExecutable(filetype, executable)

This function takes a filetype and an executable and checks whether it is safe to execute the given executable. For security reasons users may not want to have Vim execute random executables or may have forbidden to do so for specific filetypes by setting the "<filetype>_exec" variable (plugin_exec).

It returns TRUE or FALSE to indicate whether the plugin should run the given executable. It takes the following arguments:

argument	type
filetype	string
executable	string

Command-line expressions highlighting

expr-highlight

Expressions entered by the user in i_CTRL-R_=, c_CTRL-_e, quote= are highlighted by the built-in expressions parser. It uses highlight groups described in the table below, which may be overridden by colorschemes.

hl-NvimInvalid

Besides the "Nvim"-prefixed highlight groups described below, there are "NvimInvalid"-prefixed highlight groups which have the same meaning but indicate that the token contains an error or that an error occurred just before it. They have mostly the same hierarchy, except that (by default) in place of any non-Nvim-prefixed group NvimInvalid linking to Error is used and some other intermediate groups are present.

Group	Default link	Colored expression
hl-NvimInternalError	None, red/red	Parser bug
hl-NvimAssignment	Operator	Generic assignment
hl-NvimPlainAssignment	NvimAssignment	= in :let
hl-NvimAugmentedAssignment	NvimAssignment	Generic, +=/`-=`/`.=`
hl-NvimAssignmentWithAddition	NvimAugmentedAssignment	+= in :let+=

hl-NvimAssignmentWithSubtraction	NvimAugmentedAssignment	-= in :let-=
hl-NvimAssignmentWithConcatenation	NvimAugmentedAssignment	.= in :let.=
hl-NvimOperator	Operator	Generic operator
hl-NvimUnaryOperator	NvimOperator	Generic unary op
hl-NvimUnaryPlus	NvimUnaryOperator	expr-unary-+
hl-NvimUnaryMinus	NvimUnaryOperator	expr-unary--
hl-NvimNot	NvimUnaryOperator	expr-!
hl-NvimBinaryOperator	NvimOperator	Generic binary op
hl-NvimComparison	NvimBinaryOperator	Any expr4 operator
hl-NvimComparisonModifier	NvimComparison	#/`?` near expr4 op
hl-NvimBinaryPlus	NvimBinaryOperator	expr-+
hl-NvimBinaryMinus	NvimBinaryOperator	expr--
hl-NvimConcat	NvimBinaryOperator	expr-.
hl-NvimConcatOrSubscript	NvimConcat	expr-. or expr-entry
hl-NvimOr	NvimBinaryOperator	expr-barbar
hl-NvimAnd	NvimBinaryOperator	expr-&&
hl-NvimMultiplication	NvimBinaryOperator	expr-star
hl-NvimDivision	NvimBinaryOperator	expr-/
hl-NvimMod	NvimBinaryOperator	expr-%
hl-NvimTernary	NvimOperator	? in expr1
hl-NvimTernaryColon	NvimTernary	: in expr1
hl-NvimParenthesis	Delimiter	Generic bracket
hl-NvimLambda	NvimParenthesis	{/`}` in lambda
hl-NvimNestingParenthesis	NvimParenthesis	(/`)` in expr-nesting
hl-NvimCallingParenthesis	NvimParenthesis	(/`)` in expr-function
hl-NvimSubscript	NvimParenthesis	Generic subscript
hl-NvimSubscriptBracket	NvimSubscript	[/`]` in expr-[]
hl-NvimSubscriptColon	NvimSubscript	: in expr-[:]
hl-NvimCurly	NvimSubscript	{/`}` in curly-braces-names
hl-NvimContainer	NvimParenthesis	Generic container
hl-NvimDict	NvimContainer	{/`}` in dict literal
hl-NvimList	NvimContainer	[/`]` in list literal
hl-NvimIdentifier	Identifier	Generic identifier
hl-NvimIdentifierScope	NvimIdentifier	Namespace: letter before : in internal-variables
hl-NvimIdentifierScopeDelimiter	NvimIdentifier	: after namespace letter
hl-NvimIdentifierName	NvimIdentifier	Rest of the ident
hl-NvimIdentifierKey	NvimIdentifier	Identifier after expr-entry
hl-NvimColon	Delimiter	: in dict literal
hl-NvimComma	Delimiter	, in dict or list literal or expr-function
hl-NvimArrow	Delimiter	-> in lambda
hl-NvimRegister	SpecialChar	expr-register
hl-NvimNumber	Number	Non-prefix digits in integer expr-number
hl-NvimNumberPrefix	Type	0 for octal-number 0x for hex-number 0b for binary-number
hl-NvimFloat	NvimNumber	Floating-point number
hl-NvimOptionSigil	Type	& in expr-option
hl-NvimOptionScope	NvimIdentifierScope	Option scope if any
hl-NvimOptionScopeDelimiter	NvimIdentifierScopeDelimiter	: after option scope
hl-NvimOptionName	NvimIdentifier	Option name
hl-NvimEnvironmentSigil	NvimOptionSigil	$ in expr-env
hl-NvimEnvironmentName	NvimIdentifier	Env variable name
hl-NvimString	String	Generic string
hl-NvimStringBody	NvimString	Generic string literal body
hl-NvimStringQuote	NvimString	Generic string quote
hl-NvimStringSpecial	SpecialChar	Generic string non-literal body
hl-NvimSingleQuote	NvimStringQuote	' in expr-'
hl-NvimSingleQuotedBody	NvimStringBody	Literal part of expr-' string body
hl-NvimSingleQuotedQuote	NvimStringSpecial	'' inside expr-' string body
hl-NvimDoubleQuote	NvimStringQuote	" in expr-quote
hl-NvimDoubleQuotedBody	NvimStringBody	Literal part of expr-quote body
hl-NvimDoubleQuotedEscape	NvimStringSpecial	Valid expr-quote escape sequence
hl-NvimDoubleQuotedUnknownEscape	NvimInvalidValue	Unrecognized expr-quote escape sequence

Vimscript functions builtin.txt

For functions grouped by what they are used for see function-list.

1. Details vimscript-functions-details

abs({expr}) abs()

Lua: Prefer math.abs().

Return the absolute value of {expr}. When {expr} evaluates to a Float abs() returns a Float. When {expr} can be converted to a Number abs() returns a Number. Otherwise abs() gives an error message and returns -1.
Examples:

```
echo abs(1.456)
```

1.456

```
echo abs(-5.456)
```

5.456

```
echo abs(-4)
```

4

Parameters:
- {expr} (number)

Return:
(number)

acos({expr}) acos()

Lua: Prefer math.acos().

Return the arc cosine of {expr} measured in radians, as a Float in the range of [0, pi].
{expr} must evaluate to a Float or a Number in the range [-1, 1].
Returns NaN if {expr} is outside the range [-1, 1]. Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo acos(0)
```

1.570796

```
echo acos(-0.5)
```

2.094395

Parameters:
- {expr} (number)

Return:
(number)

add({object}, {expr}) add()

Lua: Prefer table.insert() for lists and .. for blobs.

Append the item {expr} to List or Blob {object}. Returns the resulting List or Blob. Examples:

```
let alist = add([1, 2, 3], item)
call add(mylist, "woodstock")
```

Note that when {expr} is a List it is appended as a single item. Use extend() to concatenate Lists.
When {object} is a Blob then {expr} must be a number.
Use insert() to add an item at another position.
Returns 1 if {object} is not a List or a Blob.

Parameters:
- {object} (any)
- {expr} (any)

Return:
(any) Resulting List or Blob, or 1 if {object} is not a List or a Blob.

and({expr}, {expr}) and()

Lua: Prefer bit.band().

Bitwise AND on the two arguments. The arguments are converted
to a number. A List, Dict or Float argument causes an error.
Also see or() and xor().
Example:

```
let flag = and(bits, 0x80)
```

Parameters:
- {expr} (number)

- {expr1} (number)

Return:
(integer)

api_info() api_info()

Returns Dictionary of api-metadata.

View it in a nice human-readable format:

```
lua vim.print(vim.fn.api_info())
```

Attributes:
api-fast

Return:
(table)

append({lnum}, {text}) append()

Lua: Prefer nvim_buf_set_lines().

When {text} is a List: Append each item of the List as a
text line below line {lnum} in the current buffer.
Otherwise append {text} as one text line below line {lnum} in
the current buffer.
Any type of item is accepted and converted to a String.
{lnum} can be zero to insert a line before the first one.
{lnum} is used like with getline().
Returns 1 for failure ({lnum} out of range or out of memory),
0 for success. When {text} is an empty list zero is returned,
no matter the value of {lnum}. Example:

```
let failed = append(line('$'), "# THE END")
let failed = append(0, ["Chapter 1", "the beginning"])
```

Parameters:
- {lnum} (integer|string)

- {text} (string|string[])

Return:
(0|1)

appendbufline({buf}, {lnum}, {text}) appendbufline()

Lua: Prefer nvim_buf_set_lines().

Like append() but append the text in buffer {buf}.

This function works only for loaded buffers. First call
bufload() if needed.

For the use of {buf}, see bufname().

{lnum} is the line number to append below. Note that using
line() would use the current buffer, not the one appending
to. Use "$" to append at the end of the buffer. Other string
values are not supported.

On success 0 is returned, on failure 1 is returned.

If {buf} is not a valid buffer or {lnum} is not valid, an
error message is given. Example:

```
let failed = appendbufline(13, 0, "# THE START")
```

However, when {text} is an empty list then no error is given
for an invalid {lnum}, since {lnum} isn't actually used.

Parameters:
- {buf} (integer|string)

- {lnum} (integer)

- {text} (string)

Return:

(0|1)

argc([{winid}]) argc()

The result is the number of files in the argument list. See arglist.
If {winid} is not supplied, the argument list of the current window is used.
If {winid} is -1, the global argument list is used.
Otherwise {winid} (a window-number or window-ID) specifies the window of which the argument list is used.
Returns -1 if the {winid} argument is invalid.

Parameters:
- {winid} (integer?)

Return:
(integer)

argidx() argidx()

The result is the current index in the argument list. 0 is the first file. argc() - 1 is the last one. See arglist.

Return:
(integer)

arglistid([{winnr} [, {tabnr}]]) arglistid()

Return the argument list ID. This is a number which identifies the argument list being used. Zero is used for the global argument list. See arglist.
Returns -1 if the arguments are invalid.

Without arguments use the current window.
With {winnr} only use this window in the current tabpage.
With {winnr} and {tabnr} use the window in the specified tab page.
{winnr} is a window-number or window-ID.

Parameters:
- {winnr} (integer?)
- {tabnr} (integer?)

Return:
(integer)

argv([{nr} [, {winid}]]) argv()

The result is the {nr}th file in the argument list. See arglist. "argv(0)" is the first one. Example:

```
let i = 0
while i < argc()
  let f = escape(fnameescape(argv(i)), '.')
  exe 'amenu Arg.' .. f .. ' :e ' .. f .. '<CR>'
  let i = i + 1
endwhile
```

Without the {nr} argument, or when {nr} is -1, a List with the whole arglist is returned.

The {winid} argument is a window-ID, see argc().
For the Vim command line arguments see v:argv.

Returns an empty string if {nr}th argument is not present in the argument list. Returns an empty List if the {winid} argument is invalid.

Parameters:
- {nr} (integer?)
- {winid} (integer?)

Return:
(string|string[])

asin({expr}) asin()

Lua: Prefer math.asin().

Return the arc sine of {expr} measured in radians, as a Float in the range of [-pi/2, pi/2].
{expr} must evaluate to a Float or a Number in the range [-1, 1].
Returns NaN if {expr} is outside the range [-1, 1]. Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo asin(0.8)
```

0.927295

```
echo asin(-0.5)
```

-0.523599

Parameters:
- {expr} (any)

Return:
(number)

assert_beeps({cmd}) assert_beeps()

Run {cmd} and add an error message to v:errors if it does NOT produce a beep or visual bell.
Also see assert_fails(), assert_nobeep() and assert-return.

Parameters:
- {cmd} (string)

Return:
(0|1)

assert_equal({expected}, {actual} [, {msg}]) assert_equal()

Lua: Prefer assert() with vim.deep_equal().

When {expected} and {actual} are not equal an error message is added to v:errors and 1 is returned. Otherwise zero is returned. assert-return
The error is in the form "Expected {expected} but got {actual}". When {msg} is present it is prefixed to that, along with the location of the assert when run from a script.

There is no automatic conversion, the String "4" is different from the Number 4. And the number 4 is different from the Float 4.0. The value of 'ignorecase' is not used here, case always matters.
Example:

```
call assert_equal('foo', 'bar', 'baz')
```

Will add the following to v:errors:

```
test.vim line 12: baz: Expected 'foo' but got 'bar'
```

Parameters:
- {expected} (any)
- {actual} (any)
- {msg} (any?)

Return:
(0|1)

assert_equalfile({fname_one}, {fname_two}) assert_equalfile()

When the files {fname_one} and {fname_two} do not contain exactly the same text an error message is added to v:errors.
Also see assert-return.
When {fname_one} or {fname_two} does not exist the error will mention that.

Parameters:
- {fname_one} (string)
- {fname_two} (string)

Return:
(0|1)

assert_exception({error} [, {msg}]) assert_exception()

When v:exception does not contain the string {error} an error message is added to v:errors. Also see assert-return.
This can be used to assert that a command throws an exception. Using the error number, followed by a colon, avoids problems with translations:

```
try
  commandthatfails
  call assert_false(1, 'command should have failed')
catch
  call assert_exception('E492:')
endtry
```

Parameters:
- {error} (any)

- {msg} (any?)

Return:
(0|1)

assert_fails() assert_fails({cmd} [, {error} [, {msg} [, {lnum} [, {context}]]]])

Run {cmd} and add an error message to v:errors if it does NOT produce an error or when {error} is not found in the error message. Also see assert-return.

When {error} is a string it must be found literally in the first reported error. Most often this will be the error code, including the colon, e.g. "E123:".

```
call assert_fails('bad cmd', 'E987:')
```

When {error} is a List with one or two strings, these are used as patterns. The first pattern is matched against the first reported error:

```
call assert_fails('cmd', ['E987:.*expected bool'])
```

The second pattern, if present, is matched against the last reported error. To only match the last error use an empty string for the first error:

```
call assert_fails('cmd', ['', 'E987:'])
```

If {msg} is empty then it is not used. Do this to get the default message when passing the {lnum} argument.

E1115

When {lnum} is present and not negative, and the {error} argument is present and matches, then this is compared with the line number at which the error was reported. That can be the line number in a function or in a script.

E1116

When {context} is present it is used as a pattern and matched against the context (script name or function name) where {lnum} is located in.

Note that beeping is not considered an error, and some failing commands only beep. Use assert_beeps() for those.

Parameters:
- {cmd} (string)

- {error} (any?)

- {msg} (any?)

- {lnum} (integer?)

- {context} (any?)

Return:
(0|1)

assert_false() assert_false({actual} [, {msg}])

Lua: Prefer assert().

When {actual} is not false an error message is added to v:errors, like with assert_equal().
The error is in the form "Expected False but got {actual}".
When {msg} is present it is prefixed to that, along with the location of the assert when run from a script.
Also see assert-return.

A value is false when it is zero. When {actual} is not a number the assert fails.

Parameters:

- {actual} (any)

- {msg} (any?)

Return:
(0|1)

assert_inrange({lower}, {upper}, {actual} [, {msg}]) assert_inrange()

Lua: Prefer assert().

This asserts number and Float values. When {actual} is lower than {lower} or higher than {upper} an error message is added to v:errors. Also see assert-return.
The error is in the form "Expected range {lower} - {upper}, but got {actual}". When {msg} is present it is prefixed to that.

Parameters:
- {lower} (number)

- {upper} (number)

- {actual} (number)

- {msg} (string?)

Return:
(0|1)

assert_match({pattern}, {actual} [, {msg}]) assert_match()

When {pattern} does not match {actual} an error message is added to v:errors. Also see assert-return.
The error is in the form "Pattern {pattern} does not match {actual}". When {msg} is present it is prefixed to that, along with the location of the assert when run from a script.

{pattern} is used as with expr-=~: The matching is always done like 'magic' was set and 'cpoptions' is empty, no matter what the actual value of 'magic' or 'cpoptions' is.

{actual} is used as a string, automatic conversion applies. Use "^" and "$" to match with the start and end of the text. Use both to match the whole text.

Example:

```
call assert_match('^f.*o$', 'foobar')
```

Will result in a string to be added to v:errors:

```
test.vim line 12: Pattern '^f.*o$' does not match 'foobar'
```

Parameters:
- {pattern} (string)

- {actual} (string)

- {msg} (string?)

Return:
(0|1)

assert_nobeep({cmd}) assert_nobeep()

Run {cmd} and add an error message to v:errors if it produces a beep or visual bell.
Also see assert_beeps().

Parameters:
- {cmd} (string)

Return:
(0|1)

assert_notequal({expected}, {actual} [, {msg}]) assert_notequal()

The opposite of assert_equal(): add an error message to v:errors when {expected} and {actual} are equal.
Also see assert-return.

Parameters:

- {expected} (any)

- {actual} (any)

- {msg} (any?)

Return:
(0|1)

assert_notmatch({pattern}, {actual} [, {msg}]) assert_notmatch()

The opposite of assert_match(): add an error message to
v:errors when {pattern} matches {actual}.
Also see assert-return.

Parameters:
- {pattern} (string)

- {actual} (string)

- {msg} (string?)

Return:
(0|1)

assert_report({msg}) assert_report()

Report a test failure directly, using String {msg}.
Always returns one.

Parameters:
- {msg} (string)

Return:
(0|1)

assert_true({actual} [, {msg}]) assert_true()

Lua: Prefer assert().

When {actual} is not true an error message is added to
v:errors, like with assert_equal().
Also see assert-return.
A value is TRUE when it is a non-zero number or v:true.
When {actual} is not a number or v:true the assert fails.
When {msg} is given it is prefixed to the default message,
along with the location of the assert when run from a script.

Parameters:
- {actual} (any)

- {msg} (string?)

Return:
(0|1)

atan({expr}) atan()

Lua: Prefer math.atan().

Return the principal value of the arc tangent of {expr}, in
the range [-pi/2, +pi/2] radians, as a Float.
{expr} must evaluate to a Float or a Number.
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo atan(100)
```

```
1.560797
```

```
echo atan(-4.01)
```

```
-1.326405
```

Parameters:
- {expr} (number)

Return:
(number)

atan2({expr1}, {expr2}) atan2()

Lua: Prefer math.atan2().

Return the arc tangent of {expr1} / {expr2}, measured in
radians, as a Float in the range [-pi, pi].
{expr1} and {expr2} must evaluate to a Float or a Number.
Returns 0.0 if {expr1} or {expr2} is not a Float or a
Number.
Examples:

```
echo atan2(-1, 1)
```

```
-0.785398
```

```
echo atan2(1, -1)
```

```
2.356194
```

Parameters:

- {expr1} (number)
- {expr2} (number)

Return:

(number)

blob2list({blob}) blob2list()

Return a List containing the number value of each byte in Blob
{blob}. Examples:

```
blob2list(0z0102.0304)  " returns [1, 2, 3, 4]
blob2list(0z)           " returns []
```

Returns an empty List on error. list2blob() does the
opposite.

Parameters:

- {blob} (any)

Return:

(any[])

browse({save}, {title}, {initdir}, {default}) browse()

Put up a file requester. This only works when "has("browse")"
returns TRUE (only in some GUI versions).
The input fields are:
    {save}   when TRUE, select file to write
    {title}   title for the requester
    {initdir}   directory to start browsing in
    {default}   default file name
An empty string is returned when the "Cancel" button is hit,
something went wrong, or browsing is not possible.

Parameters:

- {save} (any)
- {title} (string)
- {initdir} (string)
- {default} (string)

Return:

(0|1)

browsedir({title}, {initdir}) browsedir()

Put up a directory requester. This only works when
"has("browse")" returns TRUE (only in some GUI versions).
On systems where a directory browser is not supported a file
browser is used. In that case: select a file in the directory
to be used.
The input fields are:
    {title}   title for the requester
    {initdir}   directory to start browsing in
When the "Cancel" button is hit, something went wrong, or
browsing is not possible, an empty string is returned.

Parameters:

- {title} (string)
- {initdir} (string)

Return:

(0|1)

bufadd({name}) bufadd()

Adds buffer {name} to the buffer-list literally: no special chars or expansion are applied (including "~"). Returns the new (or existing matching) buffer number, or 0 on error.

The buffer is not loaded and 'buflisted' is not set. When {name} is an empty string, a new buffer is always created.

Example (Lua):

```
local buf = vim.fn.bufadd(vim.fs.normalize('someName'))
-- Set 'buflisted'; trigger BufReadPre/BufReadPost/FileType.
vim.api.nvim_buf_call(buf, vim.cmd.edit)
```

Parameters:
- {name} (string)

Return:
(integer)

bufexists({buf}) bufexists()

Lua: Prefer nvim_buf_is_valid() after resolving {buf} to a bufnr.

Checks whether a buffer with the name or number {buf} exists. Returns TRUE if the buffer exists, FALSE otherwise.
If the {buf} argument is a number, buffer numbers are used. Number zero is the alternate buffer for the current window.

If the {buf} argument is a string it must match a buffer name exactly. The name can be:
- Relative to the current directory.
- A full path.
- The name of a buffer with 'buftype' set to "nofile".
- A URL name.

Unlisted buffers will be found.
Note that help files are listed by their short name in the output of :buffers, but bufexists() requires using their long name to be able to find them.
bufexists() may report a buffer exists, but to use the name with a :buffer command you may need to use expand(). Esp for MS-Windows 8.3 names in the form "c:\DOCUME~1"
Use "bufexists(0)" to test for the existence of an alternate file name.

Parameters:
- {buf} (any)

Return:
(0|1)

buflisted({buf}) buflisted()

Lua: Prefer vim.bo or nvim_get_option_value() after resolving {buf} to a bufnr.

Checks whether a buffer called {buf} exists and is listed (has the 'buflisted' option set). Returns TRUE if so, FALSE otherwise.
The {buf} argument is used like with bufexists().

Parameters:
- {buf} (any)

Return:
(0|1)

bufload({buf}) bufload()

Ensure the buffer {buf} is loaded. When the buffer name refers to an existing file then the file is read. Otherwise the buffer will be empty. If the buffer was already loaded then there is no change. If the buffer is not related to a file then no file is read (e.g., when 'buftype' is "nofile"). If there is an existing swap file for the file of the buffer, there will be no dialog, the buffer will be loaded anyway.
The {buf} argument is used like with bufexists().

Parameters:

- {buf} (any)

bufloaded({buf}) bufloaded()

Lua: Prefer nvim_buf_is_loaded() after resolving {buf} to a bufnr.

Checks whether a buffer called {buf} exists and is loaded (shown in a window or hidden). Returns TRUE if so, FALSE otherwise.
The {buf} argument is used like with bufexists().

Parameters:
- {buf} (any)

Return:
(0|1)

bufname([{buf}]) bufname()

Lua: Prefer nvim_buf_get_name() after resolving {buf} to a bufnr.

The result is the name of a buffer. Mostly as it is displayed by the :ls command, but not using special names such as "[No Name]". If the buffer represents a directory, the name ends with a path separator, unless it was changed by :file or nvim_buf_set_name().
If {buf} is omitted the current buffer is used.
If {buf} is a Number, that buffer number's name is given. Number zero is the alternate buffer for the current window.
If {buf} is a String, it is used as a file-pattern to match with the buffer names. This is always done like 'magic' is set and 'cpoptions' is empty. When there is more than one match an empty string is returned.
"" or "%" can be used for the current buffer, "#" for the alternate buffer.
A full match is preferred, otherwise a match at the start, end or middle of the buffer name is accepted. If you only want a full match then put "^" at the start and "$" at the end of the pattern.
Listed buffers are found first. If there is a single match with a listed buffer, that one is returned. Next unlisted buffers are searched for.
If the {buf} is a String, but you want to use it as a buffer number, force it to be a Number by adding zero to it:

```
echo bufname("3" + 0)
```

If the buffer doesn't exist, or doesn't have a name, an empty string is returned.

```
echo bufname("#")       " alternate buffer name
echo bufname(3)         " name of buffer 3
echo bufname("%")       " name of current buffer
echo bufname("file2")   " name of buffer where "file2" matches.
```

Parameters:
- {buf} (integer|string?)

Return:
(string)

bufnr([{buf} [, {create}]]) bufnr()

Lua: Prefer nvim_get_current_buf() when omitted, or nvim_create_buf() with nvim_buf_set_name() for creating; name lookup, "$", "%", "#", and {create} semantics differ.

The result is the number of a buffer, as it is displayed by the :ls command. For the use of {buf}, see bufname() above.
If the buffer doesn't exist, -1 is returned. Or, if the {create} argument is present and TRUE, a new, unlisted, buffer is created and its number is returned. Example:

```
let newbuf = bufnr('Scratch001', 1)
```

Using an empty name uses the current buffer. To create a new buffer with an empty name use bufadd().

bufnr("$") is the last buffer:

```
let last_buffer = bufnr("$")
```

The result is a Number, which is the highest buffer number of existing buffers. Note that not all buffers with a smaller number necessarily exist, because ":bwipeout" may have removed them. Use bufexists() to test for the existence of a buffer.

Parameters:
- {buf} (integer|string?)

- {create} (any?)

Return:
(integer)

bufwinid({buf}) bufwinid()

Lua: Prefer nvim_tabpage_list_wins() with nvim_win_get_buf() after resolving {buf}; only the current tabpage is checked and no-match semantics differ.

Returns the window-ID of the first window associated with buffer {buf}. For the use of {buf}, see bufname() above. If buffer {buf} doesn't exist or there is no such window, -1 is returned. Example:

```
echo "A window containing buffer 1 is " .. (bufwinid(1))
```

Only deals with the current tabpage. See win_findbuf() for finding more.

Parameters:
- {buf} (any)

Return:
(integer)

bufwinnr({buf}) bufwinnr()

Lua: Prefer nvim_tabpage_list_wins() with nvim_win_get_buf() and nvim_win_get_number() after resolving {buf}; only the current tabpage is checked and no-match semantics differ.

Like bufwinid() but returns a window-number instead of a window-ID.
If buffer {buf} doesn't exist or there is no such window, -1 is returned. Example:

```
echo "A window containing buffer 1 is " .. (bufwinnr(1))
```

The number can be used with CTRL-W w and ":wincmd w" :wincmd.

Parameters:
- {buf} (any)

Return:
(integer)

byte2line({byte}) byte2line()

Return the line number that contains the character at byte count {byte} in the current buffer. This includes the end-of-line character, depending on the 'fileformat' option for the current buffer. The first character has byte count one.
Also see line2byte(), go and :goto.

Returns -1 if the {byte} value is invalid.

Parameters:
- {byte} (any)

Return:
(integer)

byteidx({expr}, {nr} [, {utf16}]) byteidx()

Lua: Prefer vim.str_byteindex() for string input with "utf-32" or "utf-16" and strict_indexing = false; composing characters and out-of-range semantics differ.

Return byte index of the {nr}th character in the String {expr}. Use zero for the first character, it then returns zero.
If there are no multibyte characters the returned value is equal to {nr}.
Composing characters are not counted separately, their byte length is added to the preceding base character. See byteidxcomp() below for counting composing characters separately.
When {utf16} is present and TRUE, {nr} is used as the UTF-16 index in the String {expr} instead of as the character index.

The UTF-16 index is the index in the string when it is encoded
with 16-bit words. If the specified UTF-16 index is in the
middle of a character (e.g. in a 4-byte character), then the
byte index of the first byte in the character is returned.
Refer to string-offset-encoding for more information.
Example :

```
echo matchstr(str, ".", byteidx(str, 3))
```

will display the fourth character. Another way to do the
same:

```
let s = strpart(str, byteidx(str, 3))
echo strpart(s, 0, byteidx(s, 1))
```

Also see strgetchar() and strcharpart().

If there are less than {nr} characters -1 is returned.
If there are exactly {nr} characters the length of the string
in bytes is returned.
See charidx() and utf16idx() for getting the character and
UTF-16 index respectively from the byte index.
Examples:

```
echo byteidx('a😊😊', 2)        " returns 5
echo byteidx('a😊😊', 2, 1)     " returns 1
echo byteidx('a😊😊', 3, 1)     " returns 5
```

Attributes:
api-fast

Parameters:
- {expr} (any)
- {nr} (integer)
- {utf16} (any?)

Return:
(integer)

byteidxcomp({expr}, {nr} [, {utf16}]) byteidxcomp()

Lua: Prefer vim.str_byteindex() with "utf-32" or
"utf-16" and strict_indexing = false.

Like byteidx(), except that a composing character is counted
as a separate character. Example:

```
let s = 'e' .. nr2char(0x301)
echo byteidx(s, 1)
echo byteidxcomp(s, 1)
echo byteidxcomp(s, 2)
```

The first and third echo result in 3 ('e' plus composing
character is 3 bytes), the second echo results in 1 ('e' is
one byte).

Attributes:
api-fast

Parameters:
- {expr} (any)
- {nr} (integer)
- {utf16} (any?)

Return:
(integer)

call({func}, {arglist} [, {dict}]) call() E699

Call function {func} with the items in List {arglist} as
arguments.
{func} can either be a Funcref or the name of a function.
a:firstline and a:lastline are set to the cursor line.
Returns the return value of the called function.
{dict} is for functions with the "dict" attribute. It will be
used to set the local variable "self". Dictionary-function

Parameters:
- {func} (any)
- {arglist} (any)

- {dict} (any?)

Return:
(any)

ceil({expr}) ceil()

Lua: Prefer math.ceil().

Return the smallest integral value greater than or equal to
{expr} as a Float (round up).
{expr} must evaluate to a Float or a Number.
Examples:

```
echo ceil(1.456)
```

2.0

```
echo ceil(-5.456)
```

-5.0

```
echo ceil(4.0)
```

4.0

Returns 0.0 if {expr} is not a Float or a Number.

Parameters:
- {expr} (number)

Return:
(number)

chanclose({id} [, {stream}]) chanclose()

Close a channel or a specific stream associated with it.
For a job, {stream} can be one of "stdin", "stdout",
"stderr" or "rpc" (closes stdin/stdout for a job started
with "rpc":v:true) If {stream} is omitted, all streams
are closed. If the channel is a pty, this will then close the
pty master, sending SIGHUP to the job process.
For a socket, there is only one stream, and {stream} should be
omitted.

Parameters:
- {id} (integer)
- {stream} (string?)

Return:
(0|1)

changenr() changenr()

Return the number of the most recent change. This is the same
number as what is displayed with :undolist and can be used
with the :undo command.
When a change was made it is the number of that change. After
redo it is the number of the redone change. After undo it is
one less than the number of the undone change.
Returns 0 if the undo list is empty.

Return:
(integer)

chansend({id}, {data}) chansend()

Lua: Prefer nvim_chan_send() for string (binary) data.

Send data to channel {id}. For a job, it writes it to the
stdin of the process. For the stdio channel channel-stdio,
it writes to Nvim's stdout. Returns the number of bytes
written if the write succeeded, 0 otherwise.
See channel-bytes for more information.

{data} may be a string, string convertible, Blob, or a list.

If {data} is a list, the items will be joined by newlines; any
newlines in an item will be sent as NUL; to send a final
newline, include a final empty string. NL-used-for-Nul

Example:

```
call chansend(id, ["abc", "123\n456", ""])
```

will send "abc<NL>123<NUL>456<NL>".

chansend() writes raw data, not RPC messages. If the channel
was created with "rpc":v:true then the channel expects RPC
messages, use rpcnotify() and rpcrequest() instead.

Parameters:

- {id} (number)

- {data} (string|string[])

Return:
(integer)

char2nr({string} [, {utf8}]) char2nr()

Lua: Prefer string.byte(): only works with ASCII.

Return Number value of the first char in {string}.
Examples:

```
echo char2nr(" ")       " returns 32
echo char2nr("ABC")     " returns 65
echo char2nr("á")       " returns 225
echo char2nr("á"[0])    " returns 195
echo char2nr("\<M-x>")  " returns 128
```

Non-ASCII characters are always treated as UTF-8 characters.
{utf8} is ignored, it exists only for backwards-compatibility.
A combining character is a separate character.
nr2char() does the opposite.

Returns 0 if {string} is not a String.

Attributes:
api-fast

Parameters:
- {string} (string)

- {utf8} (any?)

Return:
(0|1)

charclass({string}) charclass()

Return the character class of the first character in {string}.
The character class is one of:

0	blank
1	punctuation
2	word character (depends on 'iskeyword')
3	emoji
other	specific Unicode class

The class is used in patterns and word motions.
Returns 0 if {string} is not a String.

Parameters:
- {string} (string)

Return:
(0|1|2|3|'other')

charcol({expr} [, {winid}]) charcol()

Lua: Prefer nvim_win_get_cursor() with vim.str_utfindex()
for ".", or nvim_get_current_line() with
vim.str_utfindex() for "$"; marks, list input, and window
semantics differ.

Same as col() but returns the character index of the column
position given with {expr} instead of the byte position.

Example:
With the cursor on '□' in line 5 with text "□□□□":

```
echo charcol('.')       " returns 3
echo col('.')           " returns 7
```

Parameters:
- {expr} (string|any[])

- {winid} (integer?)

Return:
(integer)

charidx({string}, {idx} [, {countcc} [, {utf16}]]) charidx()

Lua: Prefer vim.str_utfindex() for string input with
"utf-32" or "utf-16" and strict_indexing = false;
countcc = false and out-of-range semantics differ.

Return the character index of the byte at {idx} in {string}.
The index of the first character is zero.

If there are no multibyte characters the returned value is
equal to {idx}.

When {countcc} is omitted or FALSE, then composing characters
are not counted separately, their byte length is added to the
preceding base character.
When {countcc} is TRUE, then composing characters are
counted as separate characters.

When {utf16} is present and TRUE, {idx} is used as the UTF-16
index in the String {expr} instead of as the byte index.

Returns -1 if the arguments are invalid or if there are less
than {idx} bytes. If there are exactly {idx} bytes the length
of the string in characters is returned.

An error is given and -1 is returned if the first argument is
not a string, the second argument is not a number or when the
third argument is present and is not zero or one.

See byteidx() and byteidxcomp() for getting the byte index
from the character index and utf16idx() for getting the
UTF-16 index from the character index.
Refer to string-offset-encoding for more information.
Examples:

```
echo charidx('áb́ć', 3)          " returns 1
echo charidx('áb́ć', 6, 1)       " returns 4
echo charidx('áb́ć', 16)         " returns -1
echo charidx('a😊😊', 4, 0, 1)  " returns 2
```

Attributes:
api-fast

Parameters:
- {string} (string)
- {idx} (integer)
- {countcc} (boolean?)
- {utf16} (boolean?)

Return:
(integer)

chdir({dir} [, {scope}]) chdir()

Sets the current-directory of the given {scope}:
- "buffer" Changes the buffer-local directory. :bcd
- "window" Changes the window-local directory. :lcd
- "tabpage" Changes the tabpage-local directory. :tcd
- "global" Changes the global directory. :cd

If {scope} is not given it is decided as follows:
- If buffer-local directory (:bcd) is set, scope is
  "buffer".
- If the window-local directory (:lcd) is set, scope is
  "window".
- If the tabpage-local directory (:tcd) is set, scope is
  "tabpage".
- Otherwise, scope is "global".

If successful, returns the previous working directory.
On failure, returns an empty string.

Example:

```
let save_dir = chdir(newdir)
if save_dir != ""
   " ... do some work
   call chdir(save_dir)
endif
```

Parameters:

- {dir} (string)

- {scope} (string?)

Return:
(string)

cindent({lnum}) cindent()

Get the amount of indent for line {lnum} according the C-indenting rules, as with 'cindent'.
The indent is counted in spaces, the value of 'tabstop' is relevant. {lnum} is used just like in getline().
When {lnum} is invalid -1 is returned.

To get or set indent of lines in a string, see vim.text.indent().

Parameters:
- {lnum} (integer|string)

Return:
(integer)

clearmatches([{win}]) clearmatches()

Clears all matches previously defined for the current window by matchadd() and the :match commands.
If {win} is specified, use this window-number or window-ID instead of the current window.

Parameters:
- {win} (integer?)

cmdcomplete_info() cmdcomplete_info()

Returns a Dictionary with information about cmdline completion. See cmdline-completion.
The items are:

cmdline_orig	The original command-line string before completion began.
pum_visible	TRUE if popup menu is visible. See pumvisible().
matches	List of all completion candidates. Each item is a string.
selected	Selected item index. First index is zero. Index is -1 if no item is selected (showing typed text only, or the last completion after no item is selected when using the <Up> or <Down> keys)

Returns an empty Dictionary if no completion was attempted, if there was only one candidate and it was fully completed, or if an error occurred.

Return:
(table<string,any>)

col({expr} [, {winid}]) col()

Lua: Prefer nvim_win_get_cursor() for "." (add 1), or Lua # on nvim_get_current_line() for "$"; marks, list input, and virtualedit differ.

Returns the byte index of the column position given with {expr}.
For accepted positions see getpos().
When {expr} is "$", it means the end of the cursor line, so the result is the number of bytes in the cursor line plus one.
Additionally {expr} can be [lnum, col]: a List with the line and column number. Most useful when the column is "$", to get the last column of a specific line. When "lnum" or "col" is out of range then col() returns zero.

With the optional {winid} argument the values are obtained for that window instead of the current window.

To get the line number use line(). To get both use getpos().

For the screen column position use virtcol(). For the character position use charcol().

Note that only marks in the current file can be used.

Examples:

```
echo col(".")                   " column of cursor
echo col("$")                   " length of cursor line plus one
echo col("'t")                  " column of mark t
echo col("'" .. markname)       " column of mark markname
```

The first column is 1. Returns 0 if {expr} is invalid or when the window with ID {winid} is not found.
For an uppercase mark the column may actually be in another buffer.
For the cursor position, when 'virtualedit' is active, the column is one higher if the cursor is after the end of the line. Also, when using a <Cmd> mapping the cursor isn't moved, this can be used to obtain the column in Insert mode:

```
imap <F2> <Cmd>echo col(".")."\n"<CR>
```

Parameters:
- {expr} (string|any[])

- {winid} (integer?)

Return:
(integer)

complete({startcol}, {matches}) complete() E785

Set the matches for Insert mode completion. Can only be used in Insert mode. Typically invoked from a mapping with CTRL-R = (see i_CTRL-R), but may also be called from a <Cmd> mapping. It does not work after CTRL-O or with an expression mapping.
{startcol} is the byte offset in the line where the completed text start. The text up to the cursor is the original text that will be replaced by the matches. Use col('.') for an empty string. "col('.') - 1" will replace one character by a match.
{matches} must be a List. Each List item is one match.
See complete-items for the kind of items that are possible.
"longest" in 'completeopt' is ignored.
Note that the after calling this function you need to avoid inserting anything that would cause completion to stop.
The match can be selected with CTRL-N and CTRL-P as usual with Insert mode completion. The popup menu will appear if specified, see ins-completion-menu.
Unlike with other ins-completion modes, the CTRL-N and CTRL-P keys can be mapped while this completion is active.
For example, to make CTRL-N move the selection without inserting the match:

```
inoremap <expr> <C-N> complete_info().mode ==# 'eval'
                        \ ? '<Down>' : '<C-N>'
```

Example:

```
inoremap <F5> <C-R>=ListMonths()<CR>

func ListMonths()
  call complete(col('.'), ['January', 'February', 'March',
        \ 'April', 'May', 'June', 'July', 'August',
        \ 'September', 'October', 'November', 'December'])
  return ''
endfunc
```

This isn't very useful, but it shows how it works. Note that an empty string is returned to avoid a zero being inserted.

Parameters:
- {startcol} (integer)

- {matches} (any[])

complete_add({expr}) complete_add()

Add {expr} to the list of matches. Only to be used by the function specified with the 'completefunc' option.
Returns 0 for failure (empty string or out of memory), 1 when the match was added, 2 when the match was already in the list.
See complete-functions for an explanation of {expr}. It is the same as one item in the list that 'omnifunc' would return.

Parameters:
- {expr} (any)

Return:
(0|1|2)

complete_check() complete_check()

Check for a key typed while looking for completion matches.

This is to be used when looking for matches takes some time.
Returns TRUE when searching for matches is to be aborted,
zero otherwise.
Only to be used by the function specified with the
'completefunc' option.

Return:

(0|1)

complete_info([{what}]) complete_info()

Returns a Dictionary with information about Insert mode
completion. See ins-completion.
The items are:

auto	TRUE when Vim started this completion by itself, which is what 'autocomplete' does, and FALSE when a key asked for it, such as i_CTRL-X_CTRL-O. A complete-functions function can read this to tell the two apart. Returned only when asked for in {what}.
completed	Return a dictionary containing the entries of the currently selected index item.
items	List of all completion candidates. Each item is a dictionary containing the entries "word", "abbr", "menu", "kind", "info", "abbr_hlgroup", "kind_hlgroup" and "user_data". The highlight group entries hold the name that the item was added with, or an empty string. "equal", "preselect" and "commit_chars" are included only for items that set them. See complete-items.
matches	Same as "items", but only returns items that are matching current query. If both "matches" and "items" are in "what", the returned list will still be named "items", but each item will have an additional "match" field.
mode	Current completion mode name string. See complete_info_mode for the values.
preinserted_text	The actual text that is pre-inserted, see preinserted().
pum_visible	TRUE if popup menu is visible. See pumvisible().
selected	Selected item index. First index is zero. Index is -1 if no item is selected (showing typed text only, or the last completion after no item is selected when using the <Up> or <Down> keys)
preview_winid	Info floating preview window id.
preview_bufnr	Info floating preview buffer id.

complete_info_mode

mode values are:

""	Not in completion mode
"keyword"	Keyword completion i_CTRL-X_CTRL-N
"ctrl_x"	Just pressed CTRL-X i_CTRL-X
"scroll"	Scrolling with i_CTRL-X_CTRL-E or i_CTRL-X_CTRL-Y
"whole_line"	Whole lines i_CTRL-X_CTRL-L
"files"	File names i_CTRL-X_CTRL-F
"tags"	Tags i_CTRL-X_CTRL-]
"path_defines"	Definition completion i_CTRL-X_CTRL-D
"path_patterns"	Include completion i_CTRL-X_CTRL-I
"dictionary"	Dictionary i_CTRL-X_CTRL-K
"thesaurus"	Thesaurus i_CTRL-X_CTRL-T
"cmdline"	Vim Command line i_CTRL-X_CTRL-V
"function"	User defined completion i_CTRL-X_CTRL-U
"omni"	Omni completion i_CTRL-X_CTRL-O
"spell"	Spelling suggestions i_CTRL-X_s
"eval"	complete() completion
"register"	Words from registers i_CTRL-X_CTRL-R
"unknown"	Other internal modes

If the optional {what} list argument is supplied, then only
the items listed in {what} are returned. Unsupported items in
{what} are silently ignored.

To get the position and size of the popup menu, see
pum_getpos(). It's also available in v:event during the
CompleteChanged event.

Returns an empty Dictionary on error.

Examples:

```
" Get all items
call complete_info()
" Get only 'mode'
call complete_info(['mode'])
" Get only 'mode' and 'pum_visible'
call complete_info(['mode', 'pum_visible'])
```

Parameters:
- {what} (any[]?)

Return:
(table)

confirm({msg} [, {choices} [, {default} [, {type}]]]) confirm()

confirm() offers the user a dialog, from which a choice can be made. It returns the number of the choice. For the first choice this is 1.

{msg} is displayed in a dialog with {choices} as the alternatives. When {choices} is missing or empty, "&OK" is used (and translated).
{msg} is a String, use '\n' to include a newline. Only on some systems the string is wrapped when it doesn't fit.

{choices} is a String, with the individual choices separated by '\n', e.g.

```
confirm("Save changes?", "&Yes\n&No\n&Cancel")
```

The letter after the '&' is the shortcut key for that choice. Thus you can type 'c' to select "Cancel". The shortcut does not need to be the first letter:

```
confirm("file has been modified", "&Save\nSave &All")
```

For the console, the first letter of each choice is used as the default shortcut key. Case is ignored.

The optional {type} String argument gives the type of dialog. It can be one of these values: "Error", "Question", "Info", "Warning" or "Generic". Only the first character is relevant. When {type} is omitted, "Generic" is used.

The optional {type} argument gives the type of dialog. This is only used for the icon of the Win32 GUI. It can be one of these values: "Error", "Question", "Info", "Warning" or "Generic". Only the first character is relevant. When {type} is omitted, "Generic" is used.

If the user aborts the dialog by pressing <Esc>, CTRL-C, or another valid interrupt key, confirm() returns 0.

An example:

```
let choice = confirm("What do you want?",
                  \ "&Apples\n&Oranges\n&Bananas", 2)
if choice == 0
     echo "make up your mind!"
elseif choice == 3
     echo "tasteful"
else
     echo "I prefer bananas myself."
endif
```

In a GUI dialog, buttons are used. The layout of the buttons depends on the 'v' flag in 'guioptions'. If it is included, the buttons are always put vertically. Otherwise, confirm() tries to put the buttons in one horizontal line. If they don't fit, a vertical layout is used anyway. For some systems the horizontal layout is always used.

Parameters:
- {msg} (string)
- {choices} (string?)
- {default} (integer?)
- {type} (string?)

Return:
(integer)

copy({expr}) copy()

Lua: Prefer vim.deepcopy(); deeper than Vimscript copy().

Make a copy of {expr}. For Numbers and Strings this isn't different from using {expr} directly.
When {expr} is a List a shallow copy is created. This means that the original List can be changed without changing the copy, and vice versa. But the items are identical, thus changing an item changes the contents of both Lists.
A Dictionary is copied in a similar way as a List.
Also see deepcopy().

Parameters:
- {expr} (T)

Return:
(T)

cos({expr}) cos()

Lua: Prefer math.cos().

Return the cosine of {expr}, measured in radians, as a Float.
{expr} must evaluate to a Float or a Number.
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo cos(100)
```

0.862319

```
echo cos(-4.01)
```

-0.646043

Parameters:
- {expr} (number)

Return:
(number)

cosh({expr}) cosh()

Lua: Prefer math.cosh().

Return the hyperbolic cosine of {expr} as a Float in the range
[1, inf].
{expr} must evaluate to a Float or a Number.
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo cosh(0.5)
```

1.127626

```
echo cosh(-0.5)
```

1.127626

Parameters:
- {expr} (number)

Return:
(number)

count({comp}, {expr} [, {ic} [, {start}]]) count() E706

Return the number of times an item with value {expr} appears
in String, List or Dictionary {comp}.

If {start} is given then start with the item with this index.
{start} can only be used with a List.

When {ic} is given and it's TRUE then case is ignored.

When {comp} is a string then the number of not overlapping
occurrences of {expr} is returned. Zero is returned when
{expr} is an empty string.

Parameters:
- {comp} (string|table|any[])
- {expr} (any)
- {ic} (boolean?)
- {start} (integer?)

Return:
(integer)

cursor({lnum}, {col} [, {off}]) cursor()

cursor({list})

Lua: Prefer nvim_win_set_cursor() with {lnum, col - 1};
off, curswant, and failure semantics differ.

Positions the cursor at the column (byte count) {col} in the
line {lnum}. The first column is one.

When there is one argument {list} this is used as a List
with two, three or four item:

[{lnum}, {col}]
[{lnum}, {col}, {off}]
[{lnum}, {col}, {off}, {curswant}]
This is like the return value of getpos() or getcurpos(), but without the first item.

To position the cursor using {col} as the character count, use setcursorcharpos().

Does not change the jumplist.
{lnum} is used like with getline(), except that if {lnum} is zero, the cursor will stay in the current line.
If {lnum} is greater than the number of lines in the buffer, the cursor will be positioned at the last line in the buffer.
If {col} is greater than the number of bytes in the line, the cursor will be positioned at the last character in the line.
If {col} is zero, the cursor will stay in the current column.
If {curswant} is given it is used to set the preferred column for vertical movement. Otherwise {col} is used.

When 'virtualedit' is used {off} specifies the offset in screen columns from the start of the character. E.g., a position within a <Tab> or after the last character.
Returns 0 when the position could be set, -1 otherwise.

Parameters:
- {list} (integer[])

Return:
(any)

debugbreak({pid}) debugbreak()

Lua: Prefer uv.kill() with "sigtrap" (or "sigint" outside MS-Windows); exact semantics differ.

Specifically used to interrupt a program being debugged. It will cause process {pid} to get a SIGTRAP. Behavior for other processes is undefined. See terminal-debug.
(Sends a SIGINT to a process {pid} other than MS-Windows)

Returns TRUE if successfully interrupted the program.
Otherwise returns FALSE.

Parameters:
- {pid} (integer)

Return:
(any)

deepcopy({expr} [, {noref}]) deepcopy() E698

Lua: Prefer vim.deepcopy().

Make a copy of {expr}. For Numbers and Strings this isn't different from using {expr} directly.
When {expr} is a List a full copy is created. This means that the original List can be changed without changing the copy, and vice versa. When an item is a List, a copy for it is made, recursively. Thus changing an item in the copy does not change the contents of the original List.

When {noref} is omitted or zero a contained List or Dictionary is only copied once. All references point to this single copy. With {noref} set to 1 every occurrence of a List or Dictionary results in a new copy. This also means that a cyclic reference causes deepcopy() to fail.

E724

Nesting is possible up to 100 levels. When there is an item that refers back to a higher level making a deep copy with {noref} set to 1 will fail.
Also see copy().

Parameters:
- {expr} (T)
- {noref} (boolean?)

Return:
(T)

delete({fname} [, {flags}]) delete()

Lua: Prefer vim.fs.rm().

Without {flags} or with {flags} empty: Deletes the file by the name {fname}.

This also works when {fname} is a symbolic link. The symbolic link itself is deleted, not what it points to.

When {flags} is "d": Deletes the directory by the name {fname}. This fails when directory {fname} is not empty.

When {flags} is "rf": Deletes the directory by the name
{fname} and everything in it, recursively. BE CAREFUL!
Note: on MS-Windows it is not possible to delete a directory
that is being used.

The result is a Number, which is 0/false if the delete
operation was successful and -1/true when the deletion failed
or partly failed.

Parameters:

- {fname} (string)
- {flags} (string?)

Return:

(integer)

deletebufline({buf}, {first} [, {last}]) deletebufline()

Lua: Prefer nvim_buf_set_lines().

Delete lines {first} to {last} (inclusive) from buffer {buf}.
If {last} is omitted then delete line {first} only.
On success 0 is returned, on failure 1 is returned.

This function works only for loaded buffers. First call
bufload() if needed.

For the use of {buf}, see bufname() above.

{first} and {last} are used like with getline(). Note that
when using line() this refers to the current buffer. Use "$"
to refer to the last line in buffer {buf}.

Parameters:

- {buf} (integer|string)
- {first} (integer|string)
- {last} (integer|string?)

Return:

(any)

dictwatcheradd({dict}, {pattern}, {callback}) dictwatcheradd()

Adds a watcher to a dictionary. A dictionary watcher is
identified by three components:

- A dictionary({dict});
- A key pattern({pattern}).
- A function({callback}).

After this is called, every change on {dict} and on keys
matching {pattern} will result in {callback} being invoked.

For example, to watch all global variables:

```
silent! call dictwatcherdel(g:, '*', 'OnDictChanged')
function! OnDictChanged(d,k,z)
  echomsg string(a:k) string(a:z)
endfunction
call dictwatcheradd(g:, '*', 'OnDictChanged')
```

For now {pattern} only accepts very simple patterns that can
contain a "*" at the end of the string, in which case it will
match every key that begins with the substring before the "*".
That means if "*" is not the last character of {pattern}, only
keys that are exactly equal as {pattern} will be matched.

The {callback} receives three arguments:

- The dictionary being watched.
- The key which changed.
- A dictionary containing the new and old values for the key.

The type of change can be determined by examining the keys
present on the third argument:

- If contains both old and new, the key was updated.

- If it contains only new, the key was added.

- If it contains only old, the key was deleted.

This function can be used by plugins to implement options with validation and parsing logic.

Parameters:
- {dict} (table)

- {pattern} (string)

- {callback} (function)

Return:
(any)

dictwatcherdel({dict}, {pattern}, {callback}) dictwatcherdel()

Removes a watcher added with dictwatcheradd(). All three arguments must match the ones passed to dictwatcheradd() in order for the watcher to be successfully deleted.

Parameters:
- {dict} (any)

- {pattern} (string)

- {callback} (function)

Return:
(any)

did_filetype() did_filetype()

Returns TRUE when autocommands are being executed and the FileType event triggered at least once for a buffer with a non-empty 'filetype'. Can be used to avoid triggering the FileType event again in the ftdetect scripts. FileType Returns FALSE when :setf FALLBACK was used.

When editing another file, the counter is reset, thus this really checks if the FileType event has been triggered for the current buffer. This allows an autocommand that starts editing another buffer to set 'filetype' and load a syntax file.

Attributes:
api-fast

Return:
(integer)

diff_filler({lnum}) diff_filler()

Returns the number of filler lines above line {lnum}.
These are the lines that were inserted at this point in another diff'ed window. These filler lines are shown in the display but don't exist in the buffer.
{lnum} is used like with getline(). Thus "." is the current line, "'m" mark m, etc.

To get the filler lines after the last buffer line, use:

```
echo diff_filler(line('$')+1)
```

Returns 0 if the current window is not in diff mode or filler lines have been disabled using 'diffopt'.

Parameters:
- {lnum} (integer|string)

Return:
(integer)

diff_hlID({lnum}, {col}) diff_hlID()

Returns the highlight ID for diff mode at line {lnum} column {col} (byte index). When the current line does not have a diff change zero is returned.
{lnum} is used like with getline(). Thus "." is the current line, "'m" mark m, etc.
{col} is 1 for the leftmost column, {lnum} is 1 for the first line.

The highlight ID can be used with synIDattr() to obtain
syntax information about the highlighting.

Parameters:
- {lnum} (integer|string)

- {col} (integer)

Return:
(any)

digraph_get({chars}) digraph_get() E1214

Return the digraph of {chars}. This should be a string with exactly two characters. If {chars} are not just two characters, or the digraph of {chars} does not exist, an error is given and an empty string is returned.

Also see digraph_getlist().

Examples:

```
" Get a built-in digraph
echo digraph_get('00')          " Returns '∞'

" Get a user-defined digraph
call digraph_set('aa', '□')
echo digraph_get('aa')          " Returns '□'
```

Parameters:
- {chars} (string)

Return:
(string)

digraph_getlist([{listall}]) digraph_getlist()

Return a list of digraphs. If the {listall} argument is given and it is TRUE, return all digraphs, including the default digraphs. Otherwise, return only user-defined digraphs.

Also see digraph_get().

Examples:

```
" Get user-defined digraphs
echo digraph_getlist()

" Get all the digraphs, including default digraphs
echo digraph_getlist(1)
```

Parameters:
- {listall} (boolean?)

Return:
(string[][])

digraph_set({chars}, {digraph}) digraph_set()

Add digraph {chars} to the list. {chars} must be a string with two characters. {digraph} is a string with one UTF-8 encoded character. E1215
Be careful, composing characters are NOT ignored. This function is similar to :digraphs command, but useful to add digraphs start with a white space.

The function result is v:true if digraph is registered. If this fails an error message is given and v:false is returned.

If you want to define multiple digraphs at once, you can use digraph_setlist().

Example:

```
call digraph_set('  ', '□')
```

Parameters:
- {chars} (string)

- {digraph} (string)

Return:
(any)

digraph_setlist({digraphlist}) digraph_setlist()

Similar to digraph_set() but this function can add multiple digraphs at once. {digraphlist} is a list composed of lists, where each list contains two strings with {chars} and

{digraph} as in digraph_set(). E1216
Example:

```
call digraph_setlist([['aa', '□'], ['ii', '□']])
```

It is similar to the following:

```
for [chars, digraph] in [['aa', '□'], ['ii', '□']]
      call digraph_set(chars, digraph)
endfor
```

Except that the function returns after the first error,
following digraphs will not be added.

Parameters:
- {digraphlist} (table<integer,string[]>)

Return:
(any)

empty({expr}) empty()

Lua: Prefer plain Lua emptiness checks, or vim.tbl_isempty()
for tables.

Return the Number 1 if {expr} is empty, zero otherwise.
- A List or Dictionary is empty when it does not have any
  items.
- A String is empty when its length is zero.
- A Number and Float are empty when their value is zero.
- v:false and v:null are empty, v:true is not.
- A Blob is empty when its length is zero.

Parameters:
- {expr} (any)

Return:
(integer)

environ() environ()

Lua: Prefer uv.os_environ() for the full environment table;
vim.env for keyed access.

Return all of environment variables as dictionary. You can
check if an environment variable exists like this:

```
echo has_key(environ(), 'HOME')
```

Note that the variable name may be CamelCase; to ignore case
use this:

```
echo index(keys(environ()), 'HOME', 0, 1) != -1
```

Attributes:
api-fast

Return:
(any)

escape({string}, {chars}) escape()

Escape the characters in {chars} that occur in {string} with a
backslash. Example:

```
echo escape('c:\program files\vim', ' \')
```

results in:

```
c:\\program\ files\\vim
```

Also see shellescape() and fnameescape().

Attributes:
api-fast

Parameters:
- {string} (string)
- {chars} (string)

Return:
(string)

eval({string}) eval()

Evaluate {string} and return the result. Especially useful to turn the result of string() back into the original value. This works for Numbers, Floats, Strings, Blobs and composites of them. Also works for Funcrefs that refer to existing functions.

Parameters:
- {string} (string)

Return:
(any)

eventhandler() eventhandler()

Returns 1 when inside an event handler. That is that Vim got interrupted while waiting for the user to type a character, e.g., when dropping a file on Vim. This means interactive commands cannot be used. Otherwise zero is returned.

Return:
(any)

executable({expr}) executable()

This function checks if an executable with the name {expr} exists. {expr} must be the name of the program without any arguments.

executable() uses the value of $PATH and/or the normal searchpath for programs.

PATHEXT

On MS-Windows the ".exe", ".bat", etc. can optionally be included. Then the extensions in $PATHEXT are tried. Thus if "foo.exe" does not exist, "foo.exe.bat" can be found. If $PATHEXT is not set then ".com;.exe;.bat;.cmd" is used. A dot by itself can be used in $PATHEXT to try using the name without an extension. When 'shell' looks like a Unix shell, then the name is also tried without adding an extension. On MS-Windows it only checks if the file exists and is not a directory, not if it's really executable.

On MS-Windows an executable in the same directory as the Vim executable is always found (it's added to $PATH at startup).

$NoDefaultCurrentDirectoryInExePath

On MS-Windows when using cmd.exe as 'shell' an executable in Vim's current working directory is also normally found, which can be disabled by setting the $NoDefaultCurrentDirectoryInExePath environment variable. This variable is always set by Vim when executing external commands (e.g., via :!, :make, or system()) for security reasons.

The result is a Number:
1 exists
0 does not exist
exepath() can be used to get the full path of an executable.

Attributes:
api-fast

Parameters:
- {expr} (string)

Return:
(0|1)

execute({command} [, {silent}]) execute()

Lua: Prefer nvim_exec() or vim.cmd().

Execute {command} and capture its output.
If {command} is a String, returns {command} output.
If {command} is a List, returns concatenated outputs.
Line continuations in {command} are not recognized.
Examples:

```
echo execute('echon "foo"')
```

foo

```
echo execute(['echon "foo"', 'echon "bar"'])
```

foobar

The optional {silent} argument can have these values:
"" no :silent used
"silent" :silent used

"silent!" :silent! used
The default is "silent". Note that with "silent!", unlike
:redir, error messages are dropped.

To get a list of lines use split() on the result:

```
execute('args')->split("\n")
```

This function is not available in the sandbox.
Note: If nested, an outer execute() will not observe output of
the inner calls.
Note: Text attributes (highlights) are not captured.
To execute a command in another window than the current one
use win_execute().

Parameters:
- {command} (string|string[])

- {silent} (''|'silent'|'silent!'?)

Return:
(string)

exepath({expr}) exepath()

Returns the full path of {expr} if it is an executable and
given as a (partial or full) path or is found in $PATH.
Returns empty string otherwise.
If {expr} starts with "./" the current-directory is used.

Attributes:
api-fast

Parameters:
- {expr} (string)

Return:
(string)

exists({expr}) exists()

Checks whether the expression {expr} is defined.
Returns TRUE if {expr} is defined, zero otherwise.

For checking for a supported feature use has().
For checking if a file exists use filereadable().

The {expr} argument is a string, which contains one of these:

varname	internal variable (see internal-variables). Also works
dict.key	for curly-braces-names, Dictionary entries, List items, etc.
list[i]	Beware that evaluating an index may cause an error message for an invalid expression. E.g.:

```
let l = [1, 2, 3]
echo exists("l[5]")
```

0

```
echo exists("l[xx]")
```

E121: Undefined variable: xx
0

&option-name	Vim option (only checks if it exists, not if it really works)
+option-name	Vim option that works.
$ENVNAME	environment variable (could also be done by comparing with an empty string)
*funcname	built-in function (see functions) or user defined function (see user-function). Also works for a variable that is a Funcref.
:cmdname	Ex command: built-in command, user command or command modifier :command. Returns: 1 for match with start of a command 2 full match with a command 3 matches several user commands To check for a supported command always check the return value to be 2.
:2match	The :2match command.
:3match	The :3match command (but you probably should not use it, it is reserved for internal usage)
#event	autocommand defined for this event
#event#pattern	autocommand defined for this event and pattern (the pattern is taken

literally and compared to the
autocommand patterns character by
character)
#group autocommand group exists
#group#event autocommand defined for this group and
event.
#group#event#pattern
autocommand defined for this group,
event and pattern.
##event autocommand for this event is
supported.

Examples:

```
echo exists("&mouse")
echo exists("$HOSTNAME")
echo exists("*strftime")
echo exists("*s:MyFunc")
echo exists("*MyFunc")
echo exists("*v:lua.Func")
echo exists("bufcount")
echo exists(":Make")
echo exists("#CursorHold")
echo exists("#BufReadPre#*.gz")
echo exists("#filetypeindent")
echo exists("#filetypeindent#FileType")
echo exists("#filetypeindent#FileType#*")
echo exists("##ColorScheme")
```

There must be no space between the symbol (&/$/*/#) and the
name.
There must be no extra characters after the name, although in
a few cases this is ignored. That may become stricter in the
future, thus don't count on it!
Working example:

```
echo exists(":make")
```

NOT working example:

```
echo exists(":make install")
```

Note that the argument must be a string, not the name of the
variable itself. For example:

```
echo exists(bufcount)
```

This doesn't check for existence of the "bufcount" variable,
but gets the value of "bufcount", and checks if that exists.

Parameters:
- {expr} (string)

Return:
(0|1)

exp({expr}) exp()

Lua: Prefer math.exp().

Return the exponential of {expr} as a Float in the range
[0, inf].
{expr} must evaluate to a Float or a Number.
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo exp(2)
```

7.389056

```
echo exp(-1)
```

0.367879

Parameters:
- {expr} (number)

Return:
(any)

expand({string} [, {nosuf} [, {list}]]) expand()

Expand wildcards and the following special keywords in
{string}. 'wildignorecase' applies.

If {list} is given and it is TRUE, a List will be returned.
Otherwise the result is a String and when there are several
matches, they are separated by <NL> characters.

If the expansion fails, the result is an empty string. A name
for a non-existing file is not included, unless {string} does

not start with '%', '#' or '<', see below.

When {string} starts with '%', '#' or '<', the expansion is done like for the cmdline-special variables with their associated modifiers. Here is a short overview:

%	Current file name
#	Alternate file name
#n	Alternate file name n
<cfile>	File name under the cursor
<afile>	Autocmd file name
<abuf>	Autocmd buffer number (as a String!)
<amatch>	Autocmd matched name
<cexpr>	C expression under the cursor
<sfile>	**Deprecated**, use <script> or <stack>
<slnum>	Sourced script line number or function line number
<sflnum>	Script file line number, also when in a function
<SID>	"<SNR>123_" where "123" is the current script ID <SID>
<script>	Sourced script file, or script file where the current function was defined. Lua: see lua-script-location.
<stack>	Call stack
<cword>	Word under the cursor
<cWORD>	WORD under the cursor
<client>	The {clientid} of the last received message

Modifiers:

:p	Expand to full path
:h	Head (last path component removed)
:t	Tail (last path component only)
:r	Root (one extension removed)
:e	Extension only

More modifiers are supported, for the full list see filename-modifiers.

Example:

```
let &tags = expand("%:p:h") .. "/tags"
```

Note that when expanding a string that starts with '%', '#' or '<', any following text is ignored. This does NOT work:

```
let doesntwork = expand("%:h.bak")
```

Use this:

```
let doeswork = expand("%:h") .. ".bak"
```

Also note that expanding "<cfile>" and others only returns the referenced file name without further expansion. If "<cfile>" is "~/.cshrc", you need to do another expand() to have the "~/" expanded into the path of the home directory:

```
echo expand(expand("<cfile>"))
```

There cannot be white space between the variables and the following modifier. The fnamemodify() function can be used to modify normal file names.

When using '%' or '#', and the current or alternate file name is not defined, an empty string is used. Using "%:p" in a buffer with no name, results in the current directory, with a '/' added.
When 'verbose' is set then expanding '%', '#' and <> items will result in an error message if the argument cannot be expanded.

When {string} does not start with '%', '#' or '<', it is expanded like a file name is expanded on the command line. 'suffixes' and 'wildignore' are used, unless the optional {nosuf} argument is given and it is TRUE.
Names for non-existing files are included. The "**" item can be used to search in a directory tree. For example, to find all "README" files in the current directory and below:

```
echo expand("**/README")
```

expand() can also be used to expand variables and environment variables that are only known in a shell. But this can be slow, because a shell may be used to do the expansion. See expr-env-expand.
The expanded variable is still handled like a list of file names. When an environment variable cannot be expanded, it is left unchanged. Thus ":echo expand('$FOOBAR')" results in "$FOOBAR".

See glob() for finding existing files. See system() for getting the raw output of an external command.

Parameters:

- {string} (string)
- {nosuf} (boolean?)
- {list} (nil|false?)

Return:
(string)

expandcmd({string} [, {options}]) expandcmd()

Expand special items in String {string} like what is done for an Ex command such as :edit. This expands special keywords, like with expand(), and environment variables, anywhere in {string}. "~user" and "~/path" are only expanded at the start.
The expansion is done in two steps: the special keywords are evaluated first, then "~" and environment variables are expanded in the result. Thus expand('%:~') keeps the "~", while expandcmd('%:~') returns the full path.

The following items are supported in the {options} Dict argument:

errmsg	If set to TRUE, error messages are displayed if an error is encountered during expansion. By default, error messages are not displayed.

Returns the expanded string. If an error is encountered during expansion, the unmodified {string} is returned.

Example:

```
echo expandcmd('make %<.o')
```

```
make /path/runtime/doc/builtin.o
```

```
echo expandcmd('make %<.o', {'errmsg': v:true})
```

Parameters:
- {string} (string)
- {options} (table?)

Return:
(any)

extend({expr1}, {expr2} [, {expr3}]) extend()

Lua: Prefer vim.list_extend() for lists, or vim.tbl_extend() and vim.tbl_deep_extend() for dicts; insertion and keep/force/error behavior differ.

{expr1} and {expr2} must be both Lists or both Dictionaries.

If they are Lists: Append {expr2} to {expr1}.
If {expr3} is given insert the items of {expr2} before the item with index {expr3} in {expr1}. When {expr3} is zero insert before the first item. When {expr3} is equal to len({expr1}) then {expr2} is appended.
Examples:

```
echo sort(extend(mylist, [7, 5]))
call extend(mylist, [2, 3], 1)
```

When {expr1} is the same List as {expr2} then the number of items copied is equal to the original length of the List.
E.g., when {expr3} is 1 you get N new copies of the first item (where N is the original length of the List).
Use add() to concatenate one item to a list. To concatenate two lists into a new list use the + operator:

```
let newlist = [1, 2, 3] + [4, 5]
```

If they are Dictionaries:
Add all entries from {expr2} to {expr1}.
If a key exists in both {expr1} and {expr2} then {expr3} is used to decide what to do:
{expr3} = "keep": keep the value of {expr1}
{expr3} = "force": use the value of {expr2}
{expr3} = "error": give an error message E737

When {expr3} is omitted then "force" is assumed.

{expr1} is changed when {expr2} is not empty. If necessary make a copy of {expr1} first or use extendnew() to return a new List/Dictionary.
{expr2} remains unchanged.

When {expr1} is locked and {expr2} is not empty the operation
fails.
Returns {expr1}. Returns 0 on error.

Parameters:

- {expr1} (table)
- {expr2} (table)
- {expr3} (table?)

Return:

(any)

extendnew({expr1}, {expr2} [, {expr3}]) extendnew()

Lua: Prefer vim.list_extend() on a copy for lists, or
vim.tbl_extend() and vim.tbl_deep_extend() for dicts;
insertion and keep/force/error behavior differ.

Like extend() but instead of adding items to {expr1} a new
List or Dictionary is created and returned. {expr1} remains
unchanged.

Parameters:

- {expr1} (table)
- {expr2} (table)
- {expr3} (table?)

Return:

(any)

feedkeys({string} [, {mode}]) feedkeys()

Lua: Prefer nvim_feedkeys().

Characters in {string} are queued for processing as if they
come from a mapping or were typed by the user.

By default the string is added to the end of the typeahead
buffer, thus if a mapping is still being executed the
characters come after them. Use the 'i' flag to insert before
other characters, they will be executed next, before any
characters from a mapping.

The function does not wait for processing of keys contained in
{string}.

To include special keys into {string}, use double-quotes
and "\..." notation expr-quote. For example,
feedkeys("\<CR>") simulates pressing of the <Enter> key. But
feedkeys('\<CR>') pushes 5 characters.
The <Ignore> keycode may be used to exit the
wait-for-character without doing anything.

{mode} is a String, which can contain these character flags:
'm'   Remap keys. This is default. If {mode} is absent,
        keys are remapped.
'n'   Do not remap keys.
't'   Handle keys as if typed; otherwise they are handled as
        if coming from a mapping. This matters for undo,
        opening folds, etc.
'L'   Lowlevel input. Other flags are not used.
'i'   Insert the string instead of appending (see above).
'x'   Execute commands until typeahead is empty. This is
        similar to using ":normal!". You can call feedkeys()
        several times without 'x' and then one time with 'x'
        (possibly with an empty {string}) to execute all the
        typeahead. Note that when Vim ends in Insert mode it
        will behave as if <Esc> is typed, to avoid getting
        stuck, waiting for a character to be typed before the
        script continues.
        Note that if you manage to call feedkeys() while
        executing commands, thus calling it recursively, then
        all typeahead will be consumed by the last call.
'!'   When used with 'x' will not end Insert mode. Can be
        used in a test when a timer is set to exit Insert mode
        a little later. Useful for testing CursorHoldI.

Return value is always 0.

Parameters:

- {string} (string)
- {mode} (string?)

Return:
(any)

filecopy({from}, {to}) filecopy()

Lua: Prefer uv.fs_copyfile() with { excl = true }; return semantics differ.

Copy the file pointed to by the name {from} to {to}. The result is a Number, which is TRUE if the file was copied successfully, and FALSE when it failed.
If a file with name {to} already exists, it will fail.
Note that it does not handle directories (yet).

This function is not available in the sandbox.

Parameters:
- {from} (string)

- {to} (string)

Return:
(0|1)

filereadable({file}) filereadable()

Lua: Prefer uv.fs_access() with "R" and uv.fs_stat() to exclude directories.

Returns TRUE if {file} exists, can be read, and is not a directory, else FALSE.

{file} is any expression, which is used as a String. If you don't care about the file being readable you can use glob().

{file} is used as-is, you may want to expand wildcards first:

```
echo filereadable('~/.vimrc')
```

```
0
```

```
echo filereadable(expand('~/.vimrc'))
```

```
1
```

Attributes:
api-fast

Parameters:
- {file} (string)

Return:
(0|1)

filewritable({file}) filewritable()

Lua: Prefer uv.fs_access() with "W" and uv.fs_stat() if you need Vim's 2 return for directories.

Checks whether a file with the name {file} exists and can be written. Returns 1 if so. If {file} doesn't exist, or is not writable, the result is 0. If {file} is a directory, and we can write to it, the result is 2.

Attributes:
api-fast

Parameters:
- {file} (string)

Return:
(0|1)

filter({expr1}, {expr2}) filter()

Lua: Prefer vim.tbl_filter() for lists; dicts, strings, and blobs differ.

{expr1} must be a List, String, Blob or Dictionary.
For each item in {expr1} evaluate {expr2} and when the result is zero or false remove the item from the List or Dictionary. Similarly for each byte in a Blob and each character in a String.

{expr2} must be a string or Funcref.

If {expr2} is a string, inside {expr2} v:val has the value of the current item. For a Dictionary v:key has the key of the current item and for a List v:key has the index of the current item. For a Blob v:key has the index of the current byte. For a String v:key has the index of the current character.
Examples:

```
call filter(mylist, 'v:val !~ "OLD"')
```

Removes the items where "OLD" appears.

```
call filter(mydict, 'v:key >= 8')
```

Removes the items with a key below 8.

```
call filter(var, 0)
```

Removes all the items, thus clears the List or Dictionary.

Note that {expr2} is the result of expression and is then used as an expression again. Often it is good to use a literal-string to avoid having to double backslashes.

If {expr2} is a Funcref it must take two arguments:

1. the key or the index of the current item.
2. the value of the current item.

The function must return TRUE if the item should be kept. Example that keeps the odd items of a list:

```
func Odd(idx, val)
  return a:idx % 2 == 1
endfunc
call filter(mylist, function('Odd'))
```

It is shorter when using a lambda:

```
call filter(myList, {idx, val -> idx * val <= 42})
```

If you do not use "val" you can leave it out:

```
call filter(myList, {idx -> idx % 2 == 1})
```

For a List and a Dictionary the operation is done in-place. If you want it to remain unmodified make a copy first:

```
let l = filter(copy(mylist), 'v:val =~ "KEEP"')
```

Returns {expr1}, the List or Dictionary that was filtered, or a new Blob or String.
When an error is encountered while evaluating {expr2} no further items in {expr1} are processed.
When {expr2} is a Funcref errors inside a function are ignored, unless it was defined with the "abort" flag.

Parameters:

- {expr1} (string|table)
- {expr2} (string|function)

Return:

(any)

finddir({name} [, {path} [, {count}]]) finddir()

Lua: Prefer vim.fs.find() with type = 'directory'; path, count, and return semantics differ.

Find directory {name} in {path}. Supports both downwards and upwards recursive directory searches. See file-searching for the syntax of {path}.

Returns the path of the first found match. When the found directory is below the current directory a relative path is returned. Otherwise a full path is returned.
If {path} is omitted or empty then 'path' is used.

If the optional {count} is given, find {count}'s occurrence of {name} in {path} instead of the first one.
When {count} is negative return all the matches in a List.

Returns an empty string if the directory is not found.

This is quite similar to the ex-command :find.

Parameters:

- {name} (string)
- {path} (string?)
- {count} (integer?)

Return:
(string|string[])

findfile({name} [, {path} [, {count}]]) findfile()

Lua: Prefer vim.fs.find(); path, count, return semantics, and 'suffixesadd' handling differ.

Just like finddir(), but find a file instead of a directory. Uses 'suffixesadd'. Example:

```
echo findfile("tags.vim", ".;")
```

Searches from the directory of the current file upwards until it finds the file "tags.vim".

Parameters:
- {name} (string)
- {path} (string?)
- {count} (integer?)

Return:
(string|string[])

flatten({list} [, {maxdepth}]) flatten()

Lua: Prefer vim.iter(list):flatten():totable() for a new flattened list; in-place updates and maxdepth differ.

Flatten {list} up to {maxdepth} levels. Without {maxdepth} the result is a List without nesting, as if {maxdepth} is a very large number. The {list} is changed in place, use flattennew() if you do not want that.

E900

{maxdepth} means how deep in nested lists changes are made. {list} is not modified when {maxdepth} is 0. {maxdepth} must be positive number.

If there is an error the number zero is returned.

Example:

```
echo flatten([1, [2, [3, 4]], 5])
```

```
[1, 2, 3, 4, 5]
```

```
echo flatten([1, [2, [3, 4]], 5], 1)
```

```
[1, 2, [3, 4], 5]
```

Parameters:
- {list} (any[])
- {maxdepth} (integer?)

Return:
(any[]|0)

flattennew({list} [, {maxdepth}]) flattennew()

Lua: Prefer vim.iter(list):flatten():totable() for a new flattened list; maxdepth differs.

Like flatten() but first make a copy of {list}.

Parameters:
- {list} (any[])
- {maxdepth} (integer?)

Return:
(any[]|0)

float2nr({expr}) float2nr()

Lua: Prefer math.modf() (first return value).

Convert {expr} to a Number by omitting the part after the decimal point. {expr} must evaluate to a Float or a Number. Returns 0 if {expr} is not a Float or a Number. When the value of {expr} is out of range for a Number the result is truncated to 0x7fffffff or -0x7fffffff (or when 64-bit Number support is enabled, 0x7fffffffffffffff or

-0x7fffffffffffffff). NaN results in -0x80000000 (or when
64-bit Number support is enabled, -0x8000000000000000).
Examples:

```
echo float2nr(3.95)
```

```
3
```

```
echo float2nr(-23.45)
```

```
-23
```

```
echo float2nr(1.0e100)
```

```
2147483647  (or 9223372036854775807)
```

```
echo float2nr(-1.0e150)
```

```
-2147483647 (or -9223372036854775807)
```

```
echo float2nr(1.0e-100)
```

```
0
```

Parameters:
- {expr} (number)

Return:
(any)

`floor({expr})` floor()

Lua: Prefer math.floor().

Return the largest integral value less than or equal to
{expr} as a Float (round down).
{expr} must evaluate to a Float or a Number.
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo floor(1.856)
```

```
1.0
```

```
echo floor(-5.456)
```

```
-6.0
```

```
echo floor(4.0)
```

```
4.0
```

Parameters:
- {expr} (number)

Return:
(any)

`fmod({expr1}, {expr2})` fmod()

Lua: Prefer math.fmod().

Return the remainder of {expr1} / {expr2}, even if the
division is not representable. Returns {expr1} - i * {expr2}
for some integer i such that if {expr2} is non-zero, the
result has the same sign as {expr1} and magnitude less than
the magnitude of {expr2}. If {expr2} is zero, the value
returned is zero. The value returned is a Float.
{expr1} and {expr2} must evaluate to a Float or a Number.
Returns 0.0 if {expr1} or {expr2} is not a Float or a
Number.
Examples:

```
echo fmod(12.33, 1.22)
```

```
0.13
```

```
echo fmod(-12.33, 1.22)
```

```
-0.13
```

Parameters:
- {expr1} (number)

- {expr2} (number)

Return:
(any)

fnameescape({filepath}) fnameescape()

Escapes {filepath} for use as a command argument.

(**Note:** To open a literal filepath programmatically use bufadd(vim.fs.normalize(…)), see open-file.)

All characters that have a special meaning, such as '%' and '|' are escaped with a backslash. For most systems the characters escaped are:

```
\t\n *?[{`$\\%#'\"|!<
```

For systems where a backslash appears in a filename, it depends on the value of 'isfname'. A leading '+' and '>' is also escaped (special after :edit and :write). And a "-" by itself (special after :cd).
Returns an empty string on error.
Example:

```
let fname = '+some str%nge|name'
exe "edit " .. fnameescape(fname)
```

results in executing:

```
edit \+some\ str\%nge\|name
```

Attributes:
api-fast

Parameters:
- {filepath} (string)

Return:
(string)

fnamemodify({fname}, {mods}) fnamemodify()

Lua: Prefer vim.fs.dirname(), vim.fs.basename(), vim.fs.abspath(), and vim.fs.normalize() for common path modifiers; modifier coverage differs.

Modify file name {fname} according to {mods}. {mods} is a string of characters like it is used for file names on the command line. See filename-modifiers.
Example:

```
echo fnamemodify("main.c", ":p:h")
```

results in:

```
/home/user/vim/vim/src
```

If {mods} is empty or an unsupported modifier is used then {fname} is returned.
When {fname} is empty then with {mods} ":h" returns ".", so that :cd can be used with it. This is different from expand('%:h') without a buffer name, which returns an empty string.
Note: Environment variables don't work in {fname}, use expand() first then.

Attributes:
api-fast

Parameters:
- {fname} (string)
- {mods} (string)

Return:
(string)

foldclosed({lnum}) foldclosed()

Returns the first line number of the closed fold containing line {lnum}. If the line {lnum} is not in a closed fold, -1 is returned.
{lnum} is used like with getline(). Thus "." is the current line, "'m" mark m, etc.

Parameters:
- {lnum} (integer|string)

Return:
(integer)

foldclosedend({lnum}) foldclosedend()

Returns the last line number of the closed fold containing

line {lnum}. If the line {lnum} is not in a closed fold,
-1 is returned.
{lnum} is used like with getline(). Thus "." is the current
line, "'m" mark m, etc.

Parameters:
- {lnum} (integer|string)

Return:
(integer)

foldlevel({lnum}) foldlevel()

Returns the fold nesting level of line {lnum} in the current
buffer. {lnum} is used like with getline() ("." is the
current line, "'m" mark m, etc).

For nested folds the deepest level is returned. If there is
no fold at line {lnum}, zero is returned. It doesn't matter
if the folds are open or closed. When used while updating
folds (from 'foldexpr') -1 is returned for lines where folds
are still to be updated and the foldlevel is unknown. As
a special case the level of the previous line is usually
available.

Parameters:
- {lnum} (integer|string)

Return:
(integer)

foldtext() foldtext()

Returns a String, to be displayed for a closed fold. This is
the default function used for the 'foldtext' option and should
only be called from evaluating 'foldtext'. It uses the
v:foldstart, v:foldend and v:folddashes variables.
The returned string looks like this:

```
+-- 45 lines: abcdef
```

The number of leading dashes depends on the foldlevel. The
"45" is the number of lines in the fold. "abcdef" is the text
in the first non-blank line of the fold. Leading white space,
"//" or "/*" and the text from the 'foldmarker' and
'commentstring' options is removed.
When used to draw the actual foldtext, the rest of the line
will be filled with the fold char from the 'fillchars'
setting.
Returns an empty string when there is no fold.

Return:
(string)

foldtextresult({lnum}) foldtextresult()

Returns the text that is displayed for the closed fold at line
{lnum}. Evaluates 'foldtext' in the appropriate context.
When there is no closed fold at {lnum} an empty string is
returned.
{lnum} is used like with getline(). Thus "." is the current
line, "'m" mark m, etc.
Useful when exporting folded text, e.g., to HTML.

Parameters:
- {lnum} (integer|string)

Return:
(string)

foreach({expr1}, {expr2}) foreach()

Lua: Prefer pairs() or ipairs().

{expr1} must be a List, String, Blob or Dictionary.
For each item in {expr1} execute {expr2}. {expr1} is not
modified; its values may be, as with :lockvar 1. E741
See map() and filter() to modify {expr1}.

{expr2} must be a string or Funcref.

If {expr2} is a string, inside {expr2} v:val has the value
of the current item. For a Dictionary v:key has the key
of the current item and for a List v:key has the index of
the current item. For a Blob v:key has the index of the
current byte. For a String v:key has the index of the
current character.
Examples:

```
call foreach(mylist, 'let used[v:val] = v:true')
```

This records the items that are in the {expr1} list.

Note that {expr2} is the result of expression and is then used as a command. Often it is good to use a literal-string to avoid having to double backslashes.

If {expr2} is a Funcref it must take two arguments:

1. the key or the index of the current item.
2. the value of the current item.

With a lambda you don't get an error if it only accepts one argument.
If the function returns a value, it is ignored.

Returns {expr1} in all cases.
When an error is encountered while executing {expr2} no further items in {expr1} are processed.
When {expr2} is a Funcref errors inside a function are ignored, unless it was defined with the "abort" flag.

Parameters:

- {expr1} (string|table)
- {expr2} (string|function)

Return:

(string|table)

fullcommand({name}) fullcommand()

Get the full command name from a short abbreviated command name; see 20.2 for details on command abbreviations.

The string argument {name} may start with a : and can include a [range], these are skipped and not returned. Returns an empty string if a command doesn't exist or if it's ambiguous (for user-defined commands).

Note: Command validation is not performed. Results depend on Vim's internal command-specific identification rules.
Examples:

```
echo [fullcommand('s')]         |" ['substitute']
echo [fullcommand('sub')]               |" ['substitute']
echo [fullcommand(': mark word')]       |" ['mark']
echo [fullcommand(': markword')]        |" ['']
echo [fullcommand('en')]                |" ['endif']
```

Parameters:

- {name} (string)

Return:

(string)

funcref({name} [, {arglist}] [, {dict}]) funcref()

Just like function(), but the returned Funcref will lookup the function by reference, not by name. This matters when the function {name} is redefined later.

Unlike function(), {name} must be an existing user function. It only works for an autoloaded function if it has already been loaded (to avoid mistakenly loading the autoload script when only intending to use the function name, use function() instead). {name} cannot be a builtin function.
Returns 0 on error.

Parameters:

- {name} (string)
- {arglist} (any?)
- {dict} (any?)

Return:

(any)

function({name} [, {arglist}] [, {dict}]) function() partial E700 E923

Return a Funcref variable that refers to function {name}. {name} can be the name of a user defined function or an internal function.

{name} can also be a Funcref or a partial. When it is a partial the dict stored in it will be used and the {dict} argument is not allowed. E.g.:

```
let FuncWithArg = function(dict.Func, [arg])
let Broken = function(dict.Func, [arg], dict)
```

When using the Funcref the function will be found by {name}, also when it was redefined later. Use funcref() to keep the same function.

When {arglist} or {dict} is present this creates a partial. That means the argument list and/or the dictionary is stored in the Funcref and will be used when the Funcref is called.

The arguments are passed to the function in front of other arguments, but after any argument from method. Example:

```
func Callback(arg1, arg2, name)
"...
endfunc
let Partial = function('Callback', ['one', 'two'])
"...
call Partial('name')
```

Invokes the function as with:

```
call Callback('one', 'two', 'name')
```

With a method:

```
func Callback(one, two, three)
"...
endfunc
let Partial = function('Callback', ['two'])
"...
eval 'one'->Partial('three')
```

Invokes the function as with:

```
call Callback('one', 'two', 'three')
```

The function() call can be nested to add more arguments to the Funcref. The extra arguments are appended to the list of arguments. Example:

```
func Callback(arg1, arg2, name)
"...
endfunc
let Func = function('Callback', ['one'])
let Func2 = function(Func, ['two'])
"...
call Func2('name')
```

Invokes the function as with:

```
call Callback('one', 'two', 'name')
```

The Dictionary is only useful when calling a "dict" function. In that case the {dict} is passed in as "self". Example:

```
function Callback() dict
   echo "called for " .. self.name
endfunction
"...
let context = {"name": "example"}
let Func = function('Callback', context)
"...
call Func()     " will echo: called for example
```

The use of function() is not needed when there are no extra arguments, these two are equivalent, if Callback() is defined as context.Callback():

```
let Func = function('Callback', context)
let Func = context.Callback
```

The argument list and the Dictionary can be combined:

```
function Callback(arg1, count) dict
"...
endfunction
let context = {"name": "example"}
let Func = function('Callback', ['one'], context)
"...
call Func(500)
```

Invokes the function as with:

```
call context.Callback('one', 500)
```

Returns 0 on error.

Parameters:

- {name} (string)

- {arglist} (any?)

- {dict} (any?)

Return:
(any)

garbagecollect([{atexit}]) garbagecollect()

Cleanup unused Lists and Dictionaries that have circular references.

There is hardly ever a need to invoke this function, as it is automatically done when Vim runs out of memory or is waiting for the user to press a key after 'updatetime'. Items without circular references are always freed when they become unused. This is useful if you have deleted a very big List and/or Dictionary with circular references in a script that runs for a long time.

When the optional {atexit} argument is one, garbage collection will also be done when exiting Vim, if it wasn't done before. This is useful when checking for memory leaks.

The garbage collection is not done immediately but only when it's safe to perform. This is when waiting for the user to type a character.

Parameters:
- {atexit} (boolean?)

Return:
(any)

get({list}, {idx} [, {default}]) get() get()-list

Get item {idx} from List {list}. When this item is not available return {default}. Return zero when {default} is omitted.

Parameters:
- {list} (any[])

- {idx} (integer)

- {default} (any?)

Return:
(any)

get({blob}, {idx} [, {default}]) get()-blob

Get byte {idx} from Blob {blob}. When this byte is not available return {default}. Return -1 when {default} is omitted.

Parameters:
- {blob} (string)

- {idx} (integer)

- {default} (any?)

Return:
(any)

get({dict}, {key} [, {default}]) get()-dict

Get item with key {key} from Dictionary {dict}. When this item is not available return {default}. Return zero when {default} is omitted. Useful example:

```
let val = get(g:, 'var_name', 'default')
```

This gets the value of g:var_name if it exists, and uses "default" when it does not exist.

Parameters:
- {dict} (table<string,any>)

- {key} (string)

- {default} (any?)

Return:
(any)

get({func}, {what}) get()-func

Get item {what} from Funcref {func}. Possible values for {what} are:

```
"name"    The function name
"func"    The function
"dict"    The dictionary
"args"    The list with arguments
"arity"   A dictionary with information about the number of
          arguments accepted by the function (minus the
          {arglist}) with the following fields:
              required    the number of positional arguments
              optional    the number of optional arguments,
                          in addition to the required ones
              varargs     TRUE if the function accepts a
                          variable number of arguments ...
```

Note: There is no error, if the {arglist} of the Funcref contains more arguments than the Funcref expects, it's not validated.

Returns zero on error.

Parameters:
- {func} (function)

- {what} (string)

Return:
(any)

getbufinfo([{buf}]) getbufinfo()

getbufinfo([{dict}])

Get information about buffers as a List of Dictionaries.

Without an argument information about all the buffers is returned.

When the argument is a Dictionary only the buffers matching the specified criteria are returned. The following keys can be specified in {dict}:

```
buflisted       include only listed buffers.
bufloaded       include only loaded buffers.
bufmodified     include only modified buffers.
```

Otherwise, {buf} specifies a particular buffer to return information for. For the use of {buf}, see bufname() above. If the buffer is found the returned List has one item. Otherwise the result is an empty list.

Each returned List item is a dictionary with the following entries:

```
bufnr           Buffer number.
changed         TRUE if the buffer is modified.
changedtick     Number of changes made to the buffer.
command         TRUE if the buffer belongs to the
                command-line window cmdwin.
hidden          TRUE if the buffer is hidden.
lastused        Timestamp in seconds, like
                localtime(), when the buffer was
                last used.
listed          TRUE if the buffer is listed.
lnum            Line number used for the buffer when
                opened in the current window.
                Only valid if the buffer has been
                displayed in the window in the past.
                If you want the line number of the
                last known cursor position in a given
                window, use line():
```

```
echo line('.', {winid})
```

```
linecount       Number of lines in the buffer (only
                valid when loaded)
loaded          TRUE if the buffer is loaded.
name            Full path to the file in the buffer.
signs           List of signs placed in the buffer.
                Each list item is a dictionary with
                the following fields:
                    id    sign identifier
                    lnum  line number
                    name  sign name
variables       A reference to the dictionary with
                buffer-local variables.
```

windows List of window-IDs that display this buffer

Examples:

```
for buf in getbufinfo()
    echo buf.name
endfor
for buf in getbufinfo({'buflisted':1})
    if buf.changed
        " ....
    endif
endfor
```

To get buffer-local options use:

```
getbufvar({bufnr}, '&option_name')
```

Parameters:
- {dict} (vim.fn.getbufinfo.dict?)

Return:
(vim.fn.getbufinfo.ret.item[])

getbufline({buf}, {lnum} [, {end}]) getbufline()

Lua: Prefer nvim_buf_get_lines().

Return a List with the lines starting from {lnum} to {end} (inclusive) in the buffer {buf}. If {end} is omitted, a List with only the line {lnum} is returned. See getbufoneline() for only getting the line.

For the use of {buf}, see bufname() above.

For {lnum} and {end} "$" can be used for the last line of the buffer. Otherwise a number must be used.

When {lnum} is smaller than 1 or bigger than the number of lines in the buffer, an empty List is returned.

When {end} is greater than the number of lines in the buffer, it is treated as {end} is set to the number of lines in the buffer. When {end} is before {lnum} an empty List is returned.

This function works only for loaded buffers. For unloaded and non-existing buffers, an empty List is returned.

Example:

```
let lines = getbufline(bufnr("myfile"), 1, "$")
```

Parameters:
- {buf} (integer|string)
- {lnum} (integer)
- {end} (integer?)

Return:
(string[])

getbufoneline({buf}, {lnum}) getbufoneline()

Lua: Prefer nvim_buf_get_lines().

Just like getbufline() but only get one line and return it as a string.

Parameters:
- {buf} (integer|string)
- {lnum} (integer)

Return:
(string)

getbufvar({buf}, {varname} [, {def}]) getbufvar()

Lua: Prefer nvim_buf_get_var() or vim.b after resolving {buf} to a bufnr; option names use nvim_get_option_value() or vim.bo.

Gets the value of a buffer-local variable or option {varname} in buffer {buf}.

{varname} is a string:
- Name of the variable (without "b:").

- If empty, gets a Dictionary of all buffer-local variables.

- If "&", gets a Dictionary of all buffer-local options.

- If it starts with "&", gets the value of a buffer-local

option.

{buf} has the same form as in bufname().

Also works for a global or buffer-local option. But not for a global variable, window-local variable or window-local option.

When the buffer or variable doesn't exist, {def} or an empty string is returned; there is no error.

Examples:

```
let bufmodified = getbufvar(1, "&mod")
echo "todo myvar = " .. getbufvar("todo", "myvar")
```

Parameters:
- {buf} (integer|string)

- {varname} (string)

- {def} (any?)

Return:
(any)

getcellwidths() getcellwidths()

Returns a List of cell widths of character ranges overridden by setcellwidths(). The format is equal to the argument of setcellwidths(). If no character ranges have their cell widths overridden, an empty List is returned.

Return:
(any)

getchangelist([{buf}]) getchangelist()

Returns the changelist for the buffer {buf}. For the use of {buf}, see bufname() above. If buffer {buf} doesn't exist, an empty list is returned.

The returned list contains two entries: a list with the change locations and the current position in the list. Each entry in the change list is a dictionary with the following entries:

col	column number
coladd	column offset for 'virtualedit'
lnum	line number

If buffer {buf} is the current buffer, then the current position refers to the position in the list. For other buffers, it is set to the length of the list.

Parameters:
- {buf} (integer|string?)

Return:
(table[])

getchar([{expr} [, {opts}]]) getchar()

Get a single character from the user or input stream.
If {expr} is omitted or is -1, wait until a character is
available.
If {expr} is 0, only get a character when one is available.
Return zero otherwise.
If {expr} is 1, only check if a character is available, it is
not consumed. Return zero if no character available.
To always get a string, specify "number" as FALSE in {opts}.

Without {expr} and when {expr} is 0 a whole character or special key is returned. If it is a single character, the result is a Number. Use nr2char() to convert it to a String. Otherwise a String is returned with the encoded character. For a special key it's a String with a sequence of bytes starting with 0x80 (decimal: 128). This is the same value as the String "\<Key>", e.g., "\<Left>". The returned value is also a String when a modifier (shift, control, alt) was used that is not included in the character. keytrans() can also be used to convert a returned String into a readable form.

When {expr} is 0 and Esc is typed, there will be a short delay while Vim waits to see if this is the start of an escape

sequence.

When {expr} is 1 only the first byte is returned. For a one-byte character it is the character itself as a number. Use nr2char() to convert it to a String.

Use getcharmod() to obtain any additional modifiers.

The optional argument {opts} is a Dict and supports the following items:

cursor	A String specifying cursor behavior when waiting for a character. "hide": hide the cursor. "keep": keep current cursor unchanged. "msg": move cursor to message area. (default: automagically decide between "keep" and "msg")
number	If TRUE, return a Number when getting a single character. If FALSE, the return value is always converted to a String, and an empty String (instead of 0) is returned when no character is available. (default: TRUE)
simplify	If TRUE, include modifiers in the character if possible. E.g., return the same value for CTRL-I and <Tab>. If FALSE, don't include modifiers in the character. (default: TRUE)

When the user clicks a mouse button, the mouse event will be returned. The position can then be found in v:mouse_col, v:mouse_lnum, v:mouse_winid and v:mouse_win. getmousepos() can also be used. Mouse move events will be ignored.
This example positions the mouse as it would normally happen:

```
let c = getchar()
if c == "\<LeftMouse>" && v:mouse_win > 0
  exe v:mouse_win .. "wincmd w"
  exe v:mouse_lnum
  exe "normal " .. v:mouse_col .. "|"
endif
```

There is no prompt, you will somehow have to make clear to the user that a character has to be typed. The screen is not redrawn, e.g. when resizing the window.

There is no mapping for the character.
Key codes are replaced, thus when the user presses the <Del> key you get the code for the <Del> key, not the raw character sequence. Examples:

```
getchar() == "\<Del>"
getchar() == "\<S-Left>"
```

This example redefines "f" to ignore case:

```
nmap f :call FindChar()<CR>
function FindChar()
  let c = nr2char(getchar())
  while col('.') < col('$') - 1
    normal l
    if getline('.')[col('.') - 1] ==? c
      break
    endif
  endwhile
endfunction
```

Parameters:

- {expr} (-1|0|1?)
- {opts} (table?)

Return:

(integer|string)

getcharmod() getcharmod()

Returns the state of the keyboard modifiers for the last character obtained with getchar() or in another way.
These values are added together:

2	shift
4	control
8	alt (meta)
16	meta (when it's different from ALT)
32	mouse double click
64	mouse triple click

96 mouse quadruple click (== 32 + 64)
128 command (Mac) or super

Only the modifiers that have not been included in the character itself are obtained. Thus Shift-a results in "A" without a modifier. Returns 0 if no modifiers are used.

Return:
(integer)

getcharpos({expr}) getcharpos()

Same as getpos(), except the column-number is character-indexed (UTF-8) instead of byte-indexed.

If getpos() returns v:maxcol, then getcharpos() returns the index of the last character.

Example:
With the cursor on '□' in line 5 with text "□□□□":

```
getcharpos('.')         returns [0, 5, 3, 0]
getpos('.')             returns [0, 5, 7, 0]
```

Parameters:
- {expr} (string)

Return:
(integer[])

getcharsearch() getcharsearch()

Return the current character search information as a {dict} with the following entries:

char	character previously used for a character search (t, f, T, or F); empty string if no character search has been performed
forward	direction of character search; 1 for forward, 0 for backward
until	type of character search; 1 for a t or T character search, 0 for an f or F character search

This can be useful to always have ; and , search forward/backward regardless of the direction of the previous character search:

```
nnoremap <expr> ; getcharsearch().forward ? ';' : ','
nnoremap <expr> , getcharsearch().forward ? ',' : ';'
```

Also see setcharsearch().

Return:
({ char: string, forward: 1|0, until: 1|0 })

getcharstr([{expr} [, {opts}]]) getcharstr()

The same as getchar(), except that this always returns a String, and "number" isn't allowed in {opts}.

Parameters:
- {expr} (-1|0|1?)
- {opts} (table?)

Return:
(string)

getcmdcomplpat() getcmdcomplpat()

Return completion pattern of the current command-line. Only works when the command line is being edited, thus requires use of c_CTRL-_e or c_CTRL-R_=. Also see getcmdtype(), setcmdpos(), getcmdline(), getcmdprompt(), getcmdcompltype() and setcmdline(). Returns an empty string when completion is not defined.

Return:
(string)

getcmdcompltype() getcmdcompltype()

Return the type of the current command-line completion. Only works when the command line is being edited, thus requires use of c_CTRL-_e or c_CTRL-R_=. See :command-completion for the return string. Also see getcmdtype(), setcmdpos(), getcmdline(), getcmdprompt(), getcmdcomplpat() and setcmdline(). Returns an empty string when completion is not defined.

To get the type of the command-line completion for a specified string, use getcompletiontype().

Return:

(string)

getcmdline() getcmdline()

Return the current command-line input. Only works when the command line is being edited, thus requires use of c_CTRL-_e or c_CTRL-R_=.
Example:

```
cmap <F7> <C-\>eescape(getcmdline(), ' \')<CR>
```

Also see getcmdtype(), getcmdpos(), setcmdpos(), getcmdprompt() and setcmdline().
Returns an empty string when entering a password or using inputsecret().

Return:
(string)

getcmdpos() getcmdpos()

Return the position of the cursor in the command line as a byte count. The first column is 1.
Only works when editing the command line, thus requires use of c_CTRL-_e or c_CTRL-R_= or an expression mapping.
Returns 0 otherwise.
Also see getcmdtype(), setcmdpos(), getcmdline(), getcmdprompt() and setcmdline().

Return:
(integer)

getcmdprompt() getcmdprompt()

Return the current command-line prompt when using functions like input() or confirm().
Only works when the command line is being edited, thus requires use of c_CTRL-_e or c_CTRL-R_=.
Also see getcmdtype(), getcmdline(), getcmdpos(), setcmdpos() and setcmdline().

Return:
(string)

getcmdscreenpos() getcmdscreenpos()

Return the screen position of the cursor in the command line as a byte count. The first column is 1.
Instead of getcmdpos(), it adds the prompt position.
Only works when editing the command line, thus requires use of c_CTRL-_e or c_CTRL-R_= or an expression mapping.
Returns 0 otherwise.
Also see getcmdpos(), setcmdpos(), getcmdline() and setcmdline().

Return:
(integer)

getcmdtype() getcmdtype()

Return the current command-line type. Possible return values are:

:	normal Ex command
>	debug mode command debug-mode
/	forward search command
?	backward search command
@	input() command
-	:insert or :append command
=	i_CTRL-R_=

Only works when editing the command line, thus requires use of c_CTRL-_e or c_CTRL-R_= or an expression mapping.
Returns an empty string otherwise.
Also see getcmdpos(), setcmdpos() and getcmdline().

Return:
(':'|'>'|'/'|'?'|'@'|'-'|'='|'')

getcmdwintype() getcmdwintype()

Return the current command-line-window type. Possible return values are the same as getcmdtype(). Returns an empty string when not in the command-line window.

Return:
(':'|'>'|'/'|'?'|'@'|'-'|'='|'')

getcompletion({pat}, {type} [, {filtered}]) getcompletion()

Return a list of command-line completion matches. The String {type} argument specifies what for. The following completion types are supported:

arglist	file names in argument list
augroup	autocmd groups
buffer	buffer names
breakpoint	:breakadd and :breakdel suboptions
cmdline	cmdline-completion result
color	color schemes

```
command         Ex command
compiler        compilers
custom,{func}   custom completion, defined via {func}
customlist,{func} custom completion, defined via {func}
diff_buffer     :diffget and :diffput completion
dir             directory names
dir_in_path     directory names in 'cdpath'
environment     environment variable names
event           autocommand events
expression      Vim expression
file            file and directory names
file_in_path    file and directory names in 'path'
filetype        filetype names 'filetype'
filetypecmd     :filetype suboptions
function        function name
help            help subjects
highlight       highlight groups
history         :history suboptions
keymap          keyboard mappings
locale          locale names (as output of locale -a)
mapclear        buffer argument
mapping         mapping name
menu            menus
messages        :messages suboptions
option          options
packadd         optional package pack-add names
retab           :retab suboptions
runtime         :runtime completion
scriptnames     sourced script names :scriptnames
shellcmd        Shell command
shellcmdline    Shell command line with filename arguments
sign            :sign suboptions
syntax          syntax file names 'syntax'
syntime         :syntime suboptions
tag             tags
tag_listfiles   tags, file names
user            user names
var             user variables
```

If {pat} is an empty string, then all the matches are returned. Otherwise only items matching {pat} are returned. See wildcards for the use of special characters in {pat}.

If the optional {filtered} flag is set to 1, then 'wildignore' is applied to filter the results. Otherwise all the matches are returned. The 'wildignorecase' option always applies.

If the 'wildoptions' option contains "fuzzy", then fuzzy matching is used to get the completion matches. Otherwise regular expression matching is used. Thus this function follows the user preference, what happens on the command line. If you do not want this you can make 'wildoptions' empty before calling getcompletion() and restore it afterwards.

If {type} is "cmdline", then the cmdline-completion result is returned. For example, to complete the possible values after a ":call" command:

```
echo getcompletion('call ', 'cmdline')
```

If there are no matches, an empty list is returned. An invalid value for {type} produces an error.

Parameters:

- {pat} (string)
- {type} (string)
- {filtered} (boolean?)

Return:

(string[])

getcompletiontype({pat}) getcompletiontype()

Return the type of the command-line completion using {pat}. When no corresponding completion type is found, an empty string is returned.
To get the current command-line completion type, use getcmdcompltype().

Parameters:

- {pat} (string)

Return:

(string)

getcurpos([{winid}]) getcurpos()

Lua: Prefer nvim_win_get_cursor(); return shape, 0-based column, and curswant/invalid-window semantics differ.

Get the position of the cursor. This is like getpos('.'), but includes an extra "curswant" item in the list:

[0, lnum, col, off, curswant]

The "curswant" number is the preferred column when moving the cursor vertically. After $ command it will be a very large number equal to v:maxcol. Also see getcursorcharpos() and getpos().
The first "bufnum" item is always zero. The byte position of the cursor is returned in "col". To get the character position, use getcursorcharpos().

The optional {winid} (window-number or window-ID) specifies the window. The last known cursor position is returned, this may be invalid for the current value of the buffer if it is not the current window.
If {winid} is invalid a list with zeroes is returned.

This can be used to save and restore the cursor position:

```
let save_cursor = getcurpos()
MoveTheCursorAround
call setpos('.', save_cursor)
```

Note that this only works within the window. See winrestview() for restoring more state.

Parameters:
- {winid} (integer?)

Return:
([integer, integer, integer, integer, integer])

getcursorcharpos([{winid}]) getcursorcharpos()

Same as getcurpos() but the column number in the returned List is a character index instead of a byte index.

Example:
With the cursor on '□' in line 3 with text "□□□□":

```
getcursorcharpos()      " returns [0, 3, 2, 0, 3]
getcurpos()             " returns [0, 3, 4, 0, 3]
```

Parameters:
- {winid} (integer?)

Return:
(any)

getcwd([{winnr} [, {tabnr} [, {bufnr}]]]) getcwd() E5000 E5002 E5006 E5007

Without arguments, returns the effective current-directory.
With {winnr} (window-number or window-ID), {tabnr} or {bufnr} the working directory of that scope is returned, ignoring 'autochdir'.

- If {winnr} is -1: gets the tabpage directory.
- If {winnr} and {tabnr} are both -1: gets the global directory.
  - **Note:** Vim returns an empty string when {tabnr} is -1.
- If {bufnr} is given: gets the buffer-local directory.
- An argument may be -1 only if preceding args are -1. E5001 ({winnr} and {tabnr} must be -1.)

Tabs, windows and buffers are identified by their respective numbers, 0 means current tab/window/buffer. Missing {tabnr} implies 0 (missing {bufnr} does not; see below). Thus the following are equivalent:

```
getcwd(0)
getcwd(0, 0)
```

Each form reports its own scope or "wider", so e.g. {winnr} never reports a buffer-local directory. The {bufnr} form falls back to the global directory, because a buffer belongs to no particular window or tabpage.

Examples:

```
getcwd(-1, -1, 0)  " Get current buffer's directory
getcwd(-1, -1, 3)  " Get directory of buffer 3
getcwd(-1, -1, -1) " Get global directory
getcwd(-1, -1)     " Get global directory
```

Parameters:
- {winnr} (integer?)

- {tabnr} (integer?)

- {bufnr} (integer?)

Return:
(string)

getenv({name}) getenv()

Lua: Prefer vim.env.

Return the value of environment variable {name}. The {name} argument is a string, without a leading '$'. Example:

```
myHome = getenv('HOME')
```

When the variable does not exist v:null is returned. That is different from a variable set to an empty string.
See also expr-env.

Parameters:
- {name} (string)

Return:
(string)

getfontname([{name}]) getfontname()

Without an argument returns the name of the normal font being used. Like what is used for the Normal highlight group hl-Normal.
With an argument a check is done whether String {name} is a valid font name. If not then an empty string is returned.
Otherwise the actual font name is returned, or {name} if the GUI does not support obtaining the real name.
Only works when the GUI is running, thus not in your vimrc or gvimrc file. Use the GUIEnter autocommand to use this function just after the GUI has started.

Parameters:
- {name} (string?)

Return:
(string)

getfperm({fname}) getfperm()

Lua: Prefer uv.fs_stat().

Returns the file permissions of the given file {fname} as a String. If {fname} does not exist or its directory cannot be read, an empty string is returned.
The result is of the form "rwxrwxrwx", where each group of "rwx" flags represent, in turn, the permissions of the owner of the file, the group the file belongs to, and other users.
If a user does not have a given permission the flag for this is replaced with the string "-". Examples:

```
echo getfperm("/etc/passwd")
echo getfperm(expand("~/.config/nvim/init.vim"))
```

This will hopefully (from a security point of view) display the string "rw-r--r--" or even "rw-------".

For setting permissions use setfperm().

Attributes:
api-fast

Parameters:
- {fname} (string)

Return:
(string)

getfsize({fname}) getfsize()

Lua: Prefer uv.fs_stat().

Returns the size in bytes of the given file {fname}.
If {fname} is a directory, 0 is returned.

If the file {fname} can't be found, -1 is returned.
If the size of {fname} is too big to fit in a Number then -2
is returned.

Attributes:
api-fast

Parameters:
- {fname} (string)

Return:
(integer)

getftime({fname}) getftime()

Lua: Prefer uv.fs_stat().

Returns the last modification time ("mtime") of file {fname}.
The value is measured as seconds since 1st Jan 1970, and may
be passed to strftime(). See also
localtime() and strftime().
If the file {fname} can't be found -1 is returned.

Attributes:
api-fast

Parameters:
- {fname} (string)

Return:
(integer)

getftype({fname}) getftype()

Lua: Prefer uv.fs_stat().

Returns a description of the type of file {fname} as a
String. If {fname} does not exist an empty string is
returned.
Here is a table over different kinds of files and their
results:

Normal file	"file"
Directory	"dir"
Symbolic link	"link"
Block device	"bdev"
Character device	"cdev"
Socket	"socket"
FIFO	"fifo"
All other	"other"

Example:

```
getftype("/home")
```

Note that a type such as "link" will only be returned on
systems that support it. On some systems only "dir" and
"file" are returned.

Attributes:
api-fast

Parameters:
- {fname} (string)

Return:
('file'|'dir'|'link'|'bdev'|'cdev'|'socket'|'fifo'|'other')

getjumplist([{winnr} [, {tabnr}]]) getjumplist()

Lua: Prefer nvim_get_context() for current jump history;
window/tab arguments and return shape differ.

Returns the jumplist for the specified window.

Without arguments use the current window.
With {winnr} (window-number or window-ID) only use this
window in the current tabpage.
With {winnr} and {tabnr} use the window in the specified tab
page. If {winnr} or {tabnr} is invalid, an empty list is
returned.

The returned list contains two entries: a list with the jump
locations and the last used jump position number in the list.
Each entry in the jump location list is a dictionary with
the following entries:

bufnr	buffer number
col	column number
coladd	column offset for 'virtualedit'
filename	filename if available
lnum	line number

Parameters:
- {winnr} (integer?)

- {tabnr} (integer?)

Return:
(vim.fn.getjumplist.ret)

getline({lnum}) getline()

Lua: Prefer nvim_get_current_line() or nvim_buf_get_lines().

Without {end} the result is a String, which is line {lnum} from the current buffer. Example:

```
getline(1)
```

When {lnum} is a String that doesn't start with a digit, line() is called to translate the String into a Number. To get the line under the cursor:

```
getline(".")
```

When {lnum} is a number smaller than 1 or bigger than the number of lines in the buffer, an empty string is returned.

Parameters:
- {lnum} (integer|string)

Return:
(string)

getline({lnum}, {end})
Lua: Prefer nvim_buf_get_lines().

When {end} is given the result is a List where each item is a line from the current buffer in the range {lnum} to {end}, including line {end}.
{end} is used in the same way as {lnum}.
Non-existing lines are silently omitted.
When {end} is before {lnum} an empty List is returned.
Example:

```
let start = line('.')
let end = search("^$") - 1
let lines = getline(start, end)
```

To get lines from another buffer see getbufline() and getbufoneline()

Parameters:
- {lnum} (integer|string)

- {end} (integer|string)

Return:
(string[])

getloclist({nr}) getloclist()

Returns a List with all the entries in the location list for window {nr} (window-number or window-ID). When {nr} is zero the current window is used.

For a location list window, the displayed location list is returned. For an invalid {nr}, an empty list is returned. Otherwise, same as getqflist().

Parameters:
- {nr} (integer)

Return:
(vim.fn.getqflist.ret.item[])

getloclist({nr}, {what})
If the optional {what} dictionary argument is supplied, then returns the items listed in {what} as a dictionary. Refer to getqflist() for the supported items in {what}.

In addition to the items supported by getqflist() in {what}, the following item is supported by getloclist():

filewinid id of the window used to display files from the location list. This field is applicable only when called from a location list window. See location-list-file-window for more details.

Returns a Dictionary with default values if there is no location list for the window {nr}.
Returns an empty Dictionary if window {nr} does not exist.

Examples (See also getqflist-examples):

```
echo getloclist(3, {'all': 0})
echo getloclist(5, {'filewinid': 0})
```

Parameters:
- {nr} (integer)

- {what} (vim.fn.getloclist.what)

Return:
(vim.fn.getloclist.ret)

getmarklist([{buf}]) getmarklist()

Without the {buf} argument returns a List with information about all the global marks. mark

If the optional {buf} argument is specified, returns the local marks defined in buffer {buf}. For the use of {buf}, see bufname(). If {buf} is invalid, an empty list is returned. For a prompt-buffer the result includes the ': mark.

Each item in the returned List is a Dict with the following:
```
    mark   name of the mark prefixed by "'"
    pos    a List with the position of the mark:
               [bufnum, lnum, col, off]
           Refer to getpos() for more information.
    file   file name
```

Refer to getpos() for getting information about a specific mark.

Parameters:
- {buf} (integer??)

Return:
(vim.fn.getmarklist.ret.item[])

getmatches([{win}]) getmatches()

Returns a List with all matches previously defined for the current window by matchadd() and the :match commands. getmatches() is useful in combination with setmatches(), as setmatches() can restore a list of matches saved by getmatches().
If {win} is specified, use this window-number or window-ID instead of the current window. If {win} is invalid, an empty list is returned.
Example:

```
echo getmatches()
```

```
[{"group": "MyGroup1", "pattern": "TODO",
"priority": 10, "id": 1}, {"group": "MyGroup2",
"pattern": "FIXME", "priority": 10, "id": 2}]
```

```
let m = getmatches()
call clearmatches()
echo getmatches()
```

```
[]
```

```
call setmatches(m)
echo getmatches()
```

```
[{"group": "MyGroup1", "pattern": "TODO",
"priority": 10, "id": 1}, {"group": "MyGroup2",
"pattern": "FIXME", "priority": 10, "id": 2}]
```

```
unlet m
```

Parameters:
- {win} (integer?)

Return:
(vim.fn.getmatches.ret.item[])

getmousepos() getmousepos()

Returns a Dictionary with the last known position of the mouse. This can be used in a mapping for a mouse click. The items are:
```
        screenrow       screen row
        screencol       screen column
        winid           Window ID of the click
```

winrow row inside "winid"
wincol column inside "winid"
line text line inside "winid"
column text column inside "winid"
coladd offset (in screen columns) from the start of the clicked char

All numbers are 1-based.

If not over a window, e.g. when in the command line, then only "screenrow" and "screencol" are valid, the others are zero.

When on the status line below a window or the vertical separator right of a window, the "line" and "column" values are zero.

When the position is after the text then "column" is the length of the text in bytes plus one.

If the mouse is over a focusable floating window then that window is used.

When using getchar() the Vim variables v:mouse_lnum, v:mouse_col and v:mouse_winid also provide these values.

Return:

(vim.fn.getmousepos.ret)

getpid() getpid()

Lua: Prefer uv.os_getpid().

Return a Number which is the process ID of the Vim process. This is a unique number, until Vim exits.

Attributes:

api-fast

Return:

(integer)

getpos({expr}) getpos()

Lua: Prefer nvim_win_get_cursor() for "." or nvim_buf_get_mark() for marks; return shape, off, and special-position semantics differ.

Gets a position, where {expr} is one of:

. Cursor position.
$ Last line in the current buffer.
'x Position of mark x (if the mark is not set, 0 is returned for all values).
w0 First line visible in current window (one if the display isn't updated, e.g. in silent-mode).
w$ Last line visible in current window (this is one less than "w0" if no lines are visible).
v End of the current Visual selection (unlike '< '> which give the previous, not current, Visual selection), or the cursor position if not in Visual mode.

To get the current selected region:

```
let region = getregionpos(getpos('v'), getpos('.'))
```

Explanation: in Visual mode "v" and "." complement each other. While "." refers to the cursor position, "v" refers to where v_o would move the cursor. So you can use "v" and "." together to get the selected region.

Note that if a mark in another file is used, the line number applies to that buffer.

The result is a List with four numbers:
[bufnum, lnum, col, off]
"bufnum" is zero, unless a mark like '0 or 'A is used, then it is the buffer number of the mark.
"lnum" and "col" are the position in the buffer. The first column is 1.
The "off" number is zero, unless 'virtualedit' is used. Then it is the offset in screen columns from the start of the character. E.g., a position within a <Tab> or after the last character.

For getting the cursor position see getcurpos().
The column number in the returned List is the byte position within the line. To get the character position in the line, use getcharpos().

The visual marks '< and '> refer to the beginning and end of the visual selection relative to the buffer. Note that this differs from setpos(), where they are relative to the cursor position.

Note that for '< and '> Visual mode matters: when it is "V" (visual line mode) the column of '< is zero and the column of '> is a large number equal to v:maxcol.
A very large column number equal to v:maxcol can be returned, in which case it means "after the end of the line".
If {expr} is invalid, returns a list with all zeros.

This can be used to save and restore the position of a mark:

```
let save_a_mark = getpos("'a")
" ...
call setpos("'a", save_a_mark)
```

Also see getcharpos(), getcurpos() and setpos().

Parameters:

- {expr} (string)

Return:

([integer, integer, integer, integer])

getqflist() getqflist()

Returns a List with all the current quickfix errors. Each list item is a dictionary with these entries:

```
bufnr   number of buffer that has the file name, use
        bufname() to get the name
module  module name
lnum    line number in the buffer (first line is 1)
end_lnum
        end of line number if the item is multiline
col     column number (first column is 1)
end_col end of column number if the item has range
vcol    TRUE: "col" is visual column
        FALSE: "col" is byte index
nr      error number
pattern search pattern used to locate the error
text    description of the error
type    type of the error, 'E', '1', etc.
valid   TRUE: recognized error message
user_data
        any type.  This entry is present only when
        user data was set for this item.
```

When there is no error list or it's empty, an empty list is returned. Quickfix list entries with a non-existing buffer number are returned with "bufnr" set to zero (**Note:** some functions accept buffer number zero for the alternate buffer, you may need to explicitly check for zero).

Useful application: Find pattern matches in multiple files and do something with them:

```
vimgrep /theword/jg *.c
for d in getqflist()
   echo bufname(d.bufnr) ':' d.lnum '=' d.text
endfor
```

Return:

(vim.fn.getqflist.ret.item[])

getqflist({what})

If the optional {what} dictionary argument is supplied, then returns only the items listed in {what} as a dictionary. The following string items are supported in {what}:

```
changedtick     get the total number of changes made
                to the list quickfix-changedtick
context get the quickfix-context
efm     errorformat to use when parsing "lines".  If
        not present, then the 'errorformat' option
        value is used.
id      get information for the quickfix list with
        quickfix-ID; zero means the id for the
        current list or the list specified by "nr"
idx     get information for the quickfix entry at this
        index in the list specified by "id" or "nr".
        If set to zero, then uses the current entry.
        See quickfix-index
items   quickfix list entries
lines   parse a list of lines using 'efm' and return
        the resulting entries.  Only a List type is
        accepted.  The current quickfix list is not
        modified.  See quickfix-parse.
nr      get information for this quickfix list; zero
        means the current quickfix list and "$" means
        the last quickfix list
qfbufnr number of the buffer displayed in the quickfix
        window.  Returns 0 if the quickfix buffer is
        not present.  See quickfix-buffer.
quickfixtextfunc
        function to get the text to display in the
        quickfix window.  Returns an empty string if
        this function is not set for the list.  See
        quickfix-window-function.
size    number of entries in the quickfix list
title   get the list title quickfix-title
winid   get the quickfix window-ID
all     all of the above quickfix properties
```

Non-string items in {what} are ignored. To get the value of a particular item, set it to zero.

If "nr" is not present then the current quickfix list is used.
If both "nr" and a non-zero "id" are specified, then the list
specified by "id" is used.
To get the number of lists in the quickfix stack, set "nr" to
"$" in {what}. The "nr" value in the returned dictionary
contains the quickfix stack size.
When "lines" is specified, all the other items except "efm"
are ignored. The returned dictionary contains the entry
"items" with the list of entries.

The returned dictionary contains the following entries:

changedtick	total number of changes made to the list quickfix-changedtick
context	quickfix list context. See quickfix-context If not present, set to "".
id	quickfix list ID quickfix-ID. If not present, set to 0.
idx	index of the quickfix entry in the list. If not present, set to 0.
items	quickfix list entries. If not present, set to an empty list.
nr	quickfix list number. If not present, set to 0
qfbufnr	number of the buffer displayed in the quickfix window. If not present, set to 0.
quickfixtextfunc	'quickfixtextfunc' setting of the list. If not present, set to "".
size	number of entries in the quickfix list. If not present, set to 0.
title	quickfix list title text. If not present, set to "".
winid	quickfix window-ID. If not present, set to 0

Examples (See also getqflist-examples):

```
echo getqflist({'all': 1})
echo getqflist({'nr': 2, 'title': 1})
echo getqflist({'lines' : ["F1:10:L10"]})
```

Parameters:
- {what} (vim.fn.getqflist.what)

Return:
(vim.fn.getqflist.ret)

getreg([{regname} [, 1 [, {list}]]]) getreg()

Returns the contents of register {regname} as a String.
Example:

```
let cliptext = getreg('*')
```

When register {regname} was not set the result is an empty
string.
The {regname} argument must be a string.

getreg('=') returns the last evaluated value of the expression
register. (For use in maps.)
getreg('=', 1) returns the expression itself, so that it can
be restored with setreg(). For other registers the extra
argument is ignored, thus you can always give it.

If {list} is present and TRUE, the result type is changed
to List. Each list item is one text line. Use it if you care
about zero bytes possibly present inside register: without
third argument both NLs and zero bytes are represented as NLs
(see NL-used-for-Nul).
When the register was not set an empty list is returned.

If {regname} is not specified, v:register is used.

Parameters:
- {regname} (string?)
- {expr} (any?)
- {list} (nil|false?)

Return:
(string)

getreginfo([{regname}]) getreginfo()

Returns detailed information about register {regname} as a
Dictionary with the following entries:

regcontents	List of lines contained in register {regname}, like getreg({regname}, 1, 1).

regtype the type of register {regname}, as in getregtype().
isunnamed Boolean flag, v:true if this register is currently pointed to by the unnamed register.
points_to for the unnamed register, gives the single letter name of the register currently pointed to (see quotequote). For example, after deleting a line with dd, this field will be "1", which is the register that got the deleted text.

The {regname} argument is a string. If {regname} is invalid or not set, an empty Dictionary will be returned.
If {regname} is not specified, v:register is used.
The returned Dictionary can be passed to setreg().

Parameters:
- {regname} (string?)

Return:
(table)

getregion({pos1}, {pos2} [, {opts}]) getregion()

Returns the list of strings from {pos1} to {pos2} from a buffer.

{pos1} and {pos2} must both be Lists with four numbers. See getpos() for the format of the list. It's possible to specify positions from a different buffer, but please note the limitations at getregion-notes.

The optional argument {opts} is a Dict and supports the following items:

type Specify the region's selection type. See getregtype() for possible values, except that the width can be omitted and an empty string cannot be used. (default: "v")

exclusive If TRUE, use exclusive selection for the end position. (default: follow 'selection')

You can get the last selection type by visualmode().
If Visual mode is active, use mode() to get the Visual mode (e.g., in a :vmap).
This function is useful to get text starting and ending in different columns, such as a charwise-visual selection.

getregion-notes

Note that:
- Order of {pos1} and {pos2} doesn't matter, it will always return content from the upper left position to the lower right position.
- If 'virtualedit' is enabled and the region is past the end of the lines, resulting lines are padded with spaces.
- If the region is blockwise and it starts or ends in the middle of a multi-cell character, it is not included but its selected part is substituted with spaces.
- If {pos1} and {pos2} are not in the same buffer, an empty list is returned.
- {pos1} and {pos2} must belong to a bufloaded() buffer.
- It is evaluated in current window context, which makes a difference if the buffer is displayed in a window with different 'virtualedit' or 'list' values.
- When specifying an exclusive selection and {pos1} and {pos2} are equal, the returned list contains a single character as if selection is inclusive, to match the behavior of an empty exclusive selection in Visual mode.

Examples:

```
xnoremap <CR>
\ <Cmd>echom getregion(
\ getpos('v'), getpos('.'), #{ type: mode() })<CR>
```

Parameters:

- {pos1} ([integer, integer, integer, integer])
- {pos2} ([integer, integer, integer, integer])
- {opts} ({type?:string, exclusive?:boolean}?)

Return:

(string[])

getregionpos({pos1}, {pos2} [, {opts}]) getregionpos()

Same as getregion(), but returns a list of positions describing the buffer text segments bound by {pos1} and {pos2}.
The segments are a pair of positions for every line:

```
[[{startpos}, {endpos}], ...]
```

The position is a List with four numbers:
    [bufnum, lnum, col, off]
"bufnum" is the buffer number.
"lnum" and "col" are the position in the buffer. The first column is 1.
If the "off" number of a starting position is non-zero, it is the offset in screen columns from the start of the character. E.g., a position within a <Tab> or after the last character.
If the "off" number of an ending position is non-zero, it is the offset of the character's first cell not included in the selection, otherwise all its cells are included.

To get the current visual selection:

```
let region = getregionpos(getpos('v'), getpos('.'))
```

The {opts} Dict supports the following items:

type	See getregion().
exclusive	See getregion().
eol	If TRUE, indicate positions beyond the end of a line with "col" values one more than the length of the line. If FALSE, positions are limited within their lines, and if a line is empty or the selection is entirely beyond the end of a line, a "col" value of 0 is used for both positions. (default: FALSE)
bounds	If TRUE, return only the outer bounds of the region as a single pair:

```
[[{startpos}, {endpos}]]
```

{startpos} is the start position on the first line of the region and {endpos} the end position on its last line. The lines in between are not visited, which is much faster for a large region.
(default: FALSE)

Using "bounds" with the same {opts} is equivalent to taking the outer positions of the full result:

```
let full = getregionpos(pos1, pos2, opts)
let bounds = [[full[0][0], full[-1][1]]]
```

When the full result is empty, e.g. because {pos1} and {pos2} are in different buffers, the result is empty as well.
Note that the two positions then come from different lines, so they describe a diagonal of the region and not its shape. For a blockwise region they are the start of the first line and the end of the last line, not the corners of the block.
Likewise, when the first line is empty and "eol" is FALSE, {startpos} has a "col" of 0 while {endpos} may not.

Parameters:

- {pos1} ([integer, integer, integer, integer])
- {pos2} ([integer, integer, integer, integer])
- {opts}

({type?:string, exclusive?:boolean, eol?:boolean, bounds?:boolean}?)

Return:

([[integer, integer, integer, integer], [integer, integer, integer, integer]][])

getregtype([{regname}]) getregtype()

Returns the type of register {regname} as a String.
The value will be one of:

"v"	for charwise text
"V"	for linewise text
"<CTRL-V>{width}"	for blockwise-visual text
""	for an empty or unknown register

<CTRL-V> is one character with value 0x16.
The {regname} argument is a string. If {regname} is not specified, v:register is used.

Parameters:

- {regname} (string?)

Return:

(string)

getscriptinfo([{opts}]) getscriptinfo()

Returns a List with information about all the sourced Vim scripts in the order they were sourced, like what :scriptnames shows.

The optional Dict argument {opts} supports the following optional items:

name	Script name match pattern. If specified, and "sid" is not specified, information about scripts with a name that match the pattern "name" are returned.
sid	Script ID <SID>. If specified, only information about the script with ID "sid" is returned and "name" is ignored.

Each item in the returned List is a Dict with the following items:

autoload	Always set to FALSE.
functions	List of script-local function names defined in the script. Present only when a particular script is specified using the "sid" item in {opts}.
name	Vim script file name.
sid	Script ID <SID>.
variables	A dictionary with the script-local variables. Present only when a particular script is specified using the "sid" item in {opts}. Note that this is a copy, the value of script-local variables cannot be changed using this dictionary.
version	Vim script version, always 1

Examples:

```
echo getscriptinfo({'name': 'myscript'})
echo getscriptinfo({'sid': 15})[0].variables
```

Parameters:

- {opts} (table?)

Return:

(vim.fn.getscriptinfo.ret[])

getstacktrace() getstacktrace()

Lua: Prefer debug.traceback().

Returns the current stack trace of Vim scripts.
Stack trace is a List, of which each item is a Dictionary with the following items:

funcref	The funcref if the stack is at a function, otherwise this item is omitted.
event	The string of the event description if the stack is at an autocmd event, otherwise this item is omitted.
lnum	The line number in the script on the stack.
filepath	The file path of the script on the stack.

Return:

(table[])

gettabinfo([{tabnr}]) gettabinfo()

If {tabnr} is not specified, then information about all the tabpages is returned as a List. Each List item is a Dictionary. Otherwise, {tabnr} specifies the tabpage number and information about that one is returned. If the tab page does not exist an empty List is returned.

Each List item is a Dictionary with the following entries:

tabnr	tabpage-number

variables a reference to the dictionary with tabpage-local variables
windows List of window-IDs in the tabpage.

Parameters:
- {tabnr} (integer?)

Return:
(any)

gettabvar({tabnr}, {varname} [, {def}]) gettabvar()

Lua: Prefer nvim_tabpage_get_var() or vim.t.

Get the value of a tab-local variable {varname} in tabpage {tabnr}. t:var
Tabs are numbered starting with one.
The {varname} argument is a string. When {varname} is empty a dictionary with all tab-local variables is returned.
Note that the name without "t:" must be used.
When the tab or variable doesn't exist {def} or an empty string is returned, there is no error message.

Parameters:
- {tabnr} (integer)
- {varname} (string)
- {def} (any?)

Return:
(any)

gettabwinvar({tabnr}, {winnr}, {varname} [, {def}]) gettabwinvar()

Lua: Prefer nvim_win_get_var() or vim.w after resolving {tabnr} and {winnr} to a winid; option names use nvim_get_option_value() or vim.wo.

Gets the value of window-local variable {varname} in {winnr} (window-number or window-ID) in tabpage-number {tabnr}.

{varname} is a string:
- Name of the variable (without "w:").
- If empty, gets a dictionary with all window-local variables.
- If "&", gets the values of all window-local options in a Dictionary.
- If it starts with "&", gets the value of a window-local option.

To get window-local variables in the current tabpage use getwinvar().

When {winnr} is zero the current window is used.

Also works for a global option, buffer-local option, window-local option, and tab-local option ('cmdheight'). But not for a global variable or buffer-local variable.

When the tab, window or variable doesn't exist, {def} or an empty string is returned; there is no error.

Examples:

```
let list_is_on = gettabwinvar(1, 2, '&list')
echo "myvar = " .. gettabwinvar(3, 1, 'myvar')
```

To get all window-local variables:

```
gettabwinvar({tabnr}, {winnr}, '&')
```

Parameters:
- {tabnr} (integer)
- {winnr} (integer)
- {varname} (string)
- {def} (any?)

Return:

(any)

gettagstack([{winnr}]) gettagstack()

Returns the tag stack of window {winnr} (window-number or window-ID) as a Dict.
When {winnr} is not specified, the current window is used.
When window {winnr} doesn't exist, an empty Dict is returned.

The returned dictionary contains the following entries:

curidx	Current index in the stack. When at top of the stack, set to (length + 1). Index of bottom of the stack is 1.
items	List of items in the stack. Each item is a dictionary containing the entries described below.
length	Number of entries in the stack.

Each item in the stack is a dictionary with the following entries:

bufnr	buffer number of the current jump
from	cursor position before the tag jump. See getpos() for the format of the returned list.
matchnr	current matching tag number. Used when multiple matching tags are found for a name.
tagname	name of the tag

See tagstack for more information about the tag stack.

Parameters:

- {winnr} (integer?)

Return:

(any)

gettext({text}) gettext()

Translate String {text} if possible.
This is mainly for use in the distributed Vim scripts. When generating message translations the {text} is extracted by xgettext, the translator can add the translated message in the .po file and Vim will lookup the translation when gettext() is called.
For {text} double quoted strings are preferred, because xgettext does not understand escaping in single quoted strings.

Parameters:

- {text} (string)

Return:

(string)

getwininfo([{winid}]) getwininfo()

Returns information about windows as a List with Dictionaries.

If {winid} is given Information about the window with that ID is returned, as a List with one item. If the window does not exist the result is an empty list.

Without {winid} information about all the windows in all the tabpages is returned.

Each List item is a Dictionary with the following entries:

botline	last complete displayed buffer line
bufnr	number of buffer in the window
height	window height (excluding winbar)
leftcol	first column displayed; only used when 'wrap' is off
loclist	1 if showing a location list
quickfix	1 if quickfix or location list window
status_height	status lines height (0 or 1)
tabnr	tabpage-number
terminal	1 if a terminal window
textoff	number of columns occupied by any 'foldcolumn', 'signcolumn' and line number in front of the text
topline	first displayed buffer line
variables	a reference to the dictionary with window-local variables
width	window width
winbar	1 if the window has a toolbar, 0 otherwise
wincol	leftmost screen column of the window; "col" from win_screenpos()
winid	window-ID
winnr	window-number
winrow	topmost screen line of the window; "row" from win_screenpos()

Parameters:
- {winid} (integer?)

Return:
(vim.fn.getwininfo.ret.item[])

getwinpos([{timeout}]) getwinpos()

Returns the [x, y] screen position of the Nvim GUI window as a List with two numbers (result of getwinposx() and getwinposy() combined).

{timeout} can be used to specify how long to wait in msec for a response. When omitted 100 msec is used.

Use a longer time for a remote terminal.
When using a value less than 10 and no response is received within that time, a previously reported position is returned, if available. This can be used to poll for the position and do some work in the meantime:

```
while 1
  let res = getwinpos(1)
  if res[0] >= 0
    break
  endif
  " Do some work here
endwhile
```

Parameters:
- {timeout} (integer?)

Return:
(any)

getwinposx() getwinposx()

Returns the X coordinate in pixels of the left hand side of the Nvim GUI window, or -1 if not available. The value can be used with :winpos.

Return:
(integer)

getwinposy() getwinposy()

Returns the Y coordinate in pixels of the top of the Nvim GUI window, or -1 if not available. The value can be used with :winpos.

Return:
(integer)

getwinvar({winnr}, {varname} [, {def}]) getwinvar()

Lua: Prefer nvim_win_get_var() or vim.w after resolving {winnr}; option names use nvim_get_option_value() or vim.wo.

Like gettabwinvar() for the current tabpage.
Examples:

```
let list_is_on = getwinvar(2, '&list')
echo "myvar = " .. getwinvar(1, 'myvar')
```

Parameters:
- {winnr} (integer)
- {varname} (string)
- {def} (any?)

Return:
(any)

glob({expr} [, {nosuf} [, {list} [, {alllinks}]]]) glob()

Expand the file wildcards in {expr}. See wildcards for the use of special characters.

Unless the optional {nosuf} argument is given and is TRUE, the 'suffixes' and 'wildignore' options apply: Names matching one of the patterns in 'wildignore' will be skipped and 'suffixes' affect the ordering of matches.
'wildignorecase' always applies.

When {list} is present and it is TRUE the result is a List with all matching files. The advantage of using a List is, you also get filenames containing newlines correctly.

Otherwise the result is a String and when there are several
matches, they are separated by <NL> characters.

If the expansion fails, the result is an empty String or List.

You can also use readdir() if you need to do complicated
things, such as limiting the number of matches.

A name for a non-existing file is not included. A symbolic
link is only included if it points to an existing file.
However, when the {alllinks} argument is present and it is
TRUE then all symbolic links are included.

For most systems backticks can be used to get files names from
any external command. Example:

```
let tagfiles = glob("`find . -name tags -print`")
let &tags = substitute(tagfiles, "\n", ",", "g")
```

The result of the program inside the backticks should be one
item per line. Spaces inside an item are allowed.

See expand() for expanding special Vim variables. See
system() for getting the raw output of an external command.

Parameters:

- {expr} (string)
- {nosuf} (boolean?)
- {list} (boolean?)
- {alllinks} (boolean?)

Return:

(any)

glob2regpat({string}) glob2regpat()

Convert a file pattern, as used by glob(), into a search
pattern. The result can be used to match with a string that
is a file name. E.g.

```
if filename =~ glob2regpat('Make*.mak')
  " ...
endif
```

This is equivalent to:

```
if filename =~ '^Make.*\.mak$'
  " ...
endif
```

When {string} is an empty string the result is "^$", match an
empty string.
Note that the result depends on the system. On MS-Windows
a backslash usually means a path separator.

Parameters:

- {string} (string)

Return:

(string)

globpath({path}, {expr} [, {nosuf} [, {list} [, {alllinks}]]]) globpath()

Perform glob() for String {expr} on all directories in {path}
and concatenate the results. Example:

```
echo globpath(&rtp, "syntax/c.vim")
```

{path} is a comma-separated list of directory names. Each
directory name is prepended to {expr} and expanded like with
glob(). A path separator is inserted when needed.
To add a comma inside a directory name escape it with a
backslash. Note that on MS-Windows a directory may have a
trailing backslash, remove it if you put a comma after it.
If the expansion fails for one of the directories, there is no
error message.

Unless the optional {nosuf} argument is given and is TRUE,
the 'suffixes' and 'wildignore' options apply: Names matching
one of the patterns in 'wildignore' will be skipped and
'suffixes' affect the ordering of matches.

When {list} is present and it is TRUE the result is a List
with all matching files. The advantage of using a List is,
you also get filenames containing newlines correctly.
Otherwise the result is a String and when there are several
matches, they are separated by <NL> characters. Example:

```
echo globpath(&rtp, "syntax/c.vim", 0, 1)
```

{allinks} is used as with glob().

The "**" item can be used to search in a directory tree.
For example, to find all "README.txt" files in the directories
in 'runtimepath' and below:

```
echo globpath(&rtp, "**/README.txt")
```

Upwards search and limiting the depth of "**" is not
supported, thus using 'path' will not always work properly.

Parameters:

- {path} (string)
- {expr} (string)
- {nosuf} (boolean?)
- {list} (boolean?)
- {allinks} (boolean?)

Return:

(any)

has({feature}) has()

Returns 1 if {feature} is supported, 0 otherwise. The
{feature} argument is a feature name like "nvim-0.2.1" or
"win32", see below. See also exists().

To get the system name use vim.uv.os_uname() in Lua:

```
print(vim.uv.os_uname().sysname)
```

If the code has a syntax error then Vimscript may skip the
rest of the line. Put :if and :endif on separate lines to
avoid the syntax error:

```
if has('feature')
  let x = this_breaks_without_the_feature()
endif
```

Vim's compile-time feature-names (prefixed with "+") are not
recognized because Nvim is always compiled with all possible
features. feature-compile

Feature names can be:

1. Nvim version. For example the "nvim-0.2.1" feature means

   that Nvim is version 0.2.1 or later:

   ```
   if has("nvim-0.2.1")
     " ...
   endif
   ```

2. Runtime condition or other pseudo-feature. For example the

   "win32" feature checks if the current system is Windows:

   ```
   if has("win32")
     " ...
   endif
   ```

feature-list

List of supported pseudo-feature names:

acl	ACL support.
android	Android system (not necessarily termux).
bsd	BSD system (not macOS, use "mac" for that).
clipboard	clipboard provider is available.
fname_case	Case in file names matters (for Darwin and MS-Windows this is not present).
gui_running	Nvim has a GUI.
hurd	GNU/Hurd system.
iconv	Can use iconv() for conversion.
linux	Linux system.
mac	MacOS system.
nvim	This is Nvim.
python3	Legacy Vim python3 interface. has-python
pythonx	Legacy Vim python_x interface. has-pythonx

sun	SunOS system.
termux	Termux, an android terminal app and packaging environment.
ttyin	input is a terminal (tty).
ttyout	output is a terminal (tty).
unix	Unix system.
vim_starting	True during startup.
win32	Windows system (32 or 64 bit).
win64	Windows system (64 bit).
wsl	WSL (Windows Subsystem for Linux) system.

has-patch

1. Vim patch. For example the "patch123" feature means that

   Vim patch 123 at the current v:version was included:

```
if v:version > 602 || v:version == 602 && has("patch148")
  " ...
endif
```

2. Vim version. For example the "patch-7.4.237" feature means

   that Nvim is Vim-compatible to version 7.4.237 or later.

```
if has("patch-7.4.237")
  " ...
endif
```

Attributes:
api-fast

Parameters:
- {feature} (string)

Return:
(0|1)

has_key({dict}, {key}) has_key()

Lua: Prefer dict[key] ~= nil.

Checks whether Dictionary {dict} has an entry with key {key}. Returns TRUE if so, FALSE otherwise. The {key} argument is a string.

Parameters:
- {dict} (table)
- {key} (string)

Return:
(0|1)

haslocaldir([{winnr} [, {tabnr} [, {bufnr}]]]) haslocaldir()

Checks whether the tabpage, window or buffer has set a local working directory. Returns 1 when the window has set a local path via :lcd, or when {winnr} is -1 and the tabpage has set a local path via :tcd, or when {winnr} and {tabnr} are -1 and {bufnr} has set a local path via :bcd, otherwise 0.

Tabs, windows and buffers are identified by their respective numbers, 0 means current tab/window/buffer. Missing {winnr} or {tabnr} implies 0 (missing {bufnr} does not; see below). Thus the following are equivalent:

```
echo haslocaldir()
echo haslocaldir(0)
echo haslocaldir(0, 0)
```

With {winnr} use that window in the current tabpage.
With {winnr} and {tabnr} use the window in that tabpage.
{winnr} is a window-number or window-ID.
If {winnr} is -1 it is ignored, only the tab is resolved.
If {bufnr} is provided, {winnr} and {tabnr} must be -1 and only the buffer is resolved. An argument may be -1 only if all preceding arguments are -1.
Examples of buffer usage:

```
haslocaldir(-1, -1, 0) " Current buf has a local directory?
haslocaldir(-1, -1, 3) " Buf #3 has a local directory?
```

Throw error if the arguments are invalid.
E5000 E5001 E5002 E5006 E5007

Parameters:

- {winnr} (integer?)
- {tabnr} (integer?)
- {bufnr} (integer?)

Return:
(0|1)

hasmapto({what} [, {mode} [, {abbr}]]) hasmapto()

Checks whether a mapping exists whose rhs contains {what}.
Returns TRUE if there is such a mapping in one of the modes
indicated by {mode}.
The arguments {what} and {mode} are strings.
When {abbr} is there and it is TRUE use abbreviations
instead of mappings. Don't forget to specify Insert and/or
Command-line mode.
Both the global mappings and the mappings local to the current
buffer are checked for a match.
If no matching mapping is found FALSE is returned.
The following characters are recognized in {mode}:

n	Normal mode
v	Visual and Select mode
x	Visual mode
s	Select mode
o	Operator-pending mode
i	Insert mode
l	Language-Argument ("r", "f", "t", etc.)
c	Command-line mode

When {mode} is omitted, "nvo" is used.

This function is useful to check if a mapping already exists
to a function in a Vim script. Example:

```
if !hasmapto('\ABCdoit')
   map <Leader>d \ABCdoit
endif
```

This installs the mapping to "\ABCdoit" only if there isn't
already a mapping to "\ABCdoit".

Parameters:
- {what} (any)
- {mode} (string?)
- {abbr} (boolean?)

Return:
(0|1)

histadd({history}, {item}) histadd()

Add the String {item} to the history {history} which can be
one of: hist-names

"cmd"	or ":"	command line history
"search"	or "/"	search pattern history
"expr"	or "="	typed expression history
"input"	or "@"	input line history
"debug"	or ">"	debug command history
empty		the current or last used history

The {history} string does not need to be the whole name, one
character is sufficient.
If {item} does already exist in the history, it will be
shifted to become the newest entry.
The result is a Number: TRUE if the operation was successful,
otherwise FALSE is returned.

Example:

```
call histadd("input", strftime("%Y %b %d"))
let date=input("Enter date: ")
```

This function is not available in the sandbox.

Parameters:
- {history} (string)
- {item} (any)

Return:
(0|1)

histdel({history} [, {item}]) histdel()

Clear {history}, i.e. delete all its entries. See hist-names for the possible values of {history}.

If the parameter {item} evaluates to a String, it is used as a regular expression. All entries matching that expression will be removed from the history (if there are any). Upper/lowercase must match, unless "\c" is used /\c. If {item} evaluates to a Number, it will be interpreted as an index, see :history-indexing. The respective entry will be removed if it exists.

The result is TRUE for a successful operation, otherwise FALSE is returned.

Examples:
Clear expression register history:

```
call histdel("expr")
```

Remove all entries starting with "*" from the search history:

```
call histdel("/", '^\*')
```

The following three are equivalent:

```
call histdel("search", histnr("search"))
call histdel("search", -1)
call histdel("search", '^' .. histget("search", -1) .. '$')
```

To delete the last search pattern and use the last-but-one for the "n" command and 'hlsearch':

```
call histdel("search", -1)
let @/ = histget("search", -1)
```

Parameters:
- {history} (string)

- {item} (any?)

Return:
(0|1)

histget({history} [, {index}]) histget()

Returns an entry from the specified command-line {history}. See hist-names for the possible values of {history}, and :history-indexing for {index}. If there is no such entry, an empty String is returned. When {index} is omitted, the most recent item from the history is used.

Examples:
Redo the second last search from history.

```
execute '/' .. histget("search", -2)
```

Define an Ex command ":H {num}" that supports re-execution of the {num}th entry from the output of :history.

```
command -nargs=1 H execute histget("cmd", 0+<args>)
```

Parameters:
- {history} (string)

- {index} (integer|string?)

Return:
(string)

histnr({history}) histnr()

The result is the Number of the current entry in {history}. See hist-names for the possible values of {history}. If an error occurred, -1 is returned.

Example:

```
let inp_index = histnr("expr")
```

Parameters:
- {history} (string)

Return:
(integer)

hlID({name}) hlID()

Lua: Prefer nvim_get_hl().

Returns the numeric ID of the highlight group with name {name}. When the highlight group doesn't exist, zero is returned.
This can be used to retrieve information about the highlight group. For example, to get the background color of the "Comment" group:

```
echo synIDattr(synIDtrans(hlID("Comment")), "bg")
```

Parameters:
- {name} (string)

Return:
(integer)

hlexists({name}) hlexists()

Lua: Prefer nvim_get_hl().

Checks whether a highlight group called {name} exists. Returns TRUE if the group has been defined in some way. Not necessarily when highlighting has been defined for it, it may also have been used for a syntax item.

Parameters:
- {name} (string|number|boolean|nil)

Return:
(0|1)

hostname() hostname()

Lua: Prefer uv.os_gethostname().

Returns the hostname of the machine on which the Nvim server (not the UI client) is currently running.

Attributes:
api-fast

Return:
(string)

iconv({string}, {from}, {to}) iconv()

Lua: Prefer vim.iconv().

Converts the encoding of {string} from {from} to {to}. Returns the converted String.
When the conversion completely fails an empty string is returned. When some characters could not be converted they are replaced with "?".
The encoding names are whatever the iconv() library function can accept, see ":!man 3 iconv".
Note that Vim uses UTF-8 for all Unicode encodings, conversion from/to UCS-2 is automatically changed to use UTF-8. You cannot use UCS-2 in a string anyway, because of the NUL bytes.

Attributes:
api-fast

Parameters:
- {string} (string)
- {from} (string)
- {to} (string)

Return:
(string)

id({expr}) id()

Returns a String which is a unique identifier of the container type (List, Dict, Blob and Partial). It is guaranteed that for the mentioned types id(v1) ==# id(v2) returns true iff type(v1) == type(v2) && v1 is v2.
Note that v:_null_string, v:_null_list, v:_null_dict and v:_null_blob have the same id() with different types because they are internally represented as NULL pointers.
id() returns a hexadecimal representation of the pointers to the containers (i.e. like 0x994a40), same as printf("%p", {expr}), but it is advised against counting on the exact format of the return value.

It is not guaranteed that id(no_longer_existing_container) will not be equal to some other id(): new containers may

reuse identifiers of the garbage-collected ones.

Parameters:
- {expr} (any)

Return:
(string)

indent({lnum}) indent()

Returns the indent of line {lnum} in the current buffer.
The indent is counted in spaces, the value of 'tabstop' is
relevant. {lnum} is used just like in
getline().
When {lnum} is invalid -1 is returned.

To get or set indent of lines in a string, see vim.text.indent().

Parameters:
- {lnum} (integer|string)

Return:
(integer)

index({object}, {expr} [, {start} [, {ic}]]) index()

Lua: Prefer vim.iter() with Iter:enumerate() and
Iter:find() for lists; result is 1-based, and start, ic,
and Blob semantics differ.

Find {expr} in {object} and return its index. See
indexof() for using a lambda to select the item.

If {object} is a List return the lowest index where the item
has a value equal to {expr}. There is no automatic
conversion, so the String "4" is different from the Number 4.
And the Number 4 is different from the Float 4.0. The value
of 'ignorecase' is not used here, case matters as indicated by
the {ic} argument.

If {object} is a Blob return the lowest index where the byte
value is equal to {expr}.

If {start} is given then start looking at the item with index
{start} (may be negative for an item relative to the end).

When {ic} is given and it is TRUE, ignore case. Otherwise
case must match.

-1 is returned when {expr} is not found in {object}.
Example:

```
let idx = index(words, "the")
if index(numbers, 123) >= 0
  " ...
endif
```

Parameters:
- {object} (any)
- {expr} (any)
- {start} (integer?)
- {ic} (boolean?)

Return:
(integer)

indexof({object}, {expr} [, {opts}]) indexof()

Lua: Prefer vim.iter() with Iter:enumerate() and
Iter:find() for list predicates; result is 1-based, and
startidx and Blob semantics differ.

Returns the index of an item in {object} where {expr} is
v:true. {object} must be a List or a Blob.

If {object} is a List, evaluate {expr} for each item in the
List until the expression is v:true and return the index of
this item.

If {object} is a Blob evaluate {expr} for each byte in the
Blob until the expression is v:true and return the index of
this byte.

{expr} must be a string or Funcref.

If {expr} is a string: If {object} is a List, inside
{expr} v:key has the index of the current List item and
v:val has the value of the item. If {object} is a Blob,

inside {expr} v:key has the index of the current byte and
v:val has the byte value.

If {expr} is a Funcref it must take two arguments:
1. the key or the index of the current item.

2. the value of the current item.

The function must return TRUE if the item is found and the
search should stop.

The optional argument {opts} is a Dict and supports the
following items:
    startidx   start evaluating {expr} at the item with this
                index; may be negative for an item relative to
                the end
Returns -1 when {expr} evaluates to v:false for all the items.
Example:

```
let l = [#{n: 10}, #{n: 20}, #{n: 30}]
echo indexof(l, "v:val.n == 20")
echo indexof(l, {i, v -> v.n == 30})
echo indexof(l, "v:val.n == 20", #{startidx: 1})
```

Parameters:
- {object} (any)

- {expr} (any)

- {opts} (table?)

Return:
(integer)

input({prompt} [, {text} [, {completion}]]) input()

Lua: Prefer vim.ui.input() for callback-based input.

Parameters:
- {prompt} (string)

- {text} (string?)

- {completion} (string?)

Return:
(string)

input({opts})

Lua: Prefer vim.ui.input() for callback-based input.

Prompts the user to enter text on the command-line, and
returns the text as a String. The {prompt} argument is either
a prompt string, or a blank string (for no prompt). A '\n'
can be used in the prompt to start a new line.

In the second form it accepts a single dictionary with the
following keys, any of which may be omitted:

Key	Default	Description
prompt	""	Same as {prompt} in the first form.
default	""	Same as {text} in the first form.
completion	nothing	Same as {completion} in the first form.
cancelreturn	""	The value returned when the dialog is cancelled.
highlight	nothing	Highlight handler: Funcref.

The highlighting set with :echohl is used for the prompt.
The input is entered just like a command-line, with the same
editing commands and mappings. There is a separate history
for lines typed for input().
Example:

```
if input("Coffee or beer? ") == "beer"
  echo "Cheers!"
endif
```

If the optional {text} argument is present and not empty, this
is used for the default reply, as if the user typed this.
Example:

```
let color = input("Color? ", "white")
```

The optional {completion} argument specifies the type of
completion supported for the input. Without it completion is
not performed. The supported completion types are the same as

that can be supplied to a user-defined command using the
"-complete=" argument. Refer to :command-completion for
more information. Example:

```
let fname = input("File: ", "", "file")
```

input()-highlight E5400 E5402

The optional highlight key allows specifying function which
will be used for highlighting user input. This function
receives user input as its only argument and must return
a list of 3-tuples [hl_start_col, hl_end_col + 1, hl_group]
where
        hl_start_col is the first highlighted column,
        hl_end_col is the last highlighted column (+ 1!),
        hl_group is :hi group used for highlighting.
E5403 E5404 E5405 E5406

Both hl_start_col and hl_end_col + 1 must point to the start
of the multibyte character (highlighting must not break
multibyte characters), hl_end_col + 1 may be equal to the
input length. Start column must be in range [0, len(input)),
end column must be in range (hl_start_col, len(input)],
sections must be ordered so that next hl_start_col is greater
then or equal to previous hl_end_col.

Example (try some input with parentheses):

```
highlight RBP1 guibg=Red ctermbg=red
highlight RBP2 guibg=Yellow ctermbg=yellow
highlight RBP3 guibg=Green ctermbg=green
highlight RBP4 guibg=Blue ctermbg=blue
let g:rainbow_levels = 4
function! RainbowParens(cmdline)
  let ret = []
  let i = 0
  let lvl = 0
  while i < len(a:cmdline)
    if a:cmdline[i] is# '('
      call add(ret, [i, i + 1, 'RBP' .. ((lvl % g:rainbow_levels) + 1)])
      let lvl += 1
    elseif a:cmdline[i] is# ')'
      let lvl -= 1
      call add(ret, [i, i + 1, 'RBP' .. ((lvl % g:rainbow_levels) + 1)])
    endif
    let i += 1
  endwhile
  return ret
endfunction
call input({'prompt':'>','highlight':'RainbowParens'})
```

Highlight function is called at least once for each new
displayed input string, before command-line is redrawn. It is
expected that function is pure for the duration of one input()
call, i.e. it produces the same output for the same input, so
output may be memoized. Function is run like under :silent
modifier. If the function causes any errors, it will be
skipped for the duration of the current input() call.

Highlighting is disabled if command-line contains arabic
characters.

NOTE: This function must not be used in a startup file, for
the versions that only run in GUI mode (e.g., the Win32 GUI).
Note: When input() is called from within a mapping it will
consume remaining characters from that mapping, because a
mapping is handled like the characters were typed.
Use inputsave() before input() and inputrestore()
after input() to avoid that. Another solution is to avoid
that further characters follow in the mapping, e.g., by using
:execute or :normal.

Example with a mapping:

```
nmap \x :call GetFoo()<CR>:exe "/" .. Foo<CR>
function GetFoo()
  call inputsave()
  let g:Foo = input("enter search pattern: ")
  call inputrestore()
endfunction
```

Parameters:
- {opts} (table)

Return:
(string)

inputlist({textlist}) inputlist()

Lua: Prefer vim.ui.select() for simple selection menus;
callback, returned item, and out-of-range semantics differ.

Displays a list of strings and prompts the user to select
one by entering a number. {textlist} must be a List of

strings. Returns the number the user entered.
The user can also select an item by clicking on it with the
mouse, if the mouse is enabled in the command line ('mouse' is
"a" or includes "c"). For the first string 0 is returned.
When clicking above the first item a negative number is
returned. When clicking on the prompt one more than the
length of {textlist} is returned.
Make sure {textlist} has less than 'lines' entries, otherwise
it won't work. It's a good idea to put the entry number at
the start of the string. And put a prompt in the first item.
Example:

```
let color = inputlist(['Select color:', '1. red',
        \ '2. green', '3. blue'])
```

Parameters:
- {textlist} (string[])

Return:
(any)

inputrestore() inputrestore()

Restore typeahead that was saved with a previous inputsave().
Should be called the same number of times inputsave() is
called. Calling it more often is harmless though.
Returns TRUE when there is nothing to restore, FALSE
otherwise.

Return:
(integer)

inputsave() inputsave()

Preserve typeahead (also from mappings) and clear it, so that
a following prompt gets input from the user. Should be
followed by a matching inputrestore() after the prompt. Can
be used several times, in which case there must be just as
many inputrestore() calls.
Returns TRUE when out of memory, FALSE otherwise.

Return:
(integer)

inputsecret({prompt} [, {text}]) inputsecret()

This function acts much like the input() function with but
two exceptions:
a) the user's response will be displayed as a sequence of
asterisks ("*") thereby keeping the entry secret, and
b) the user's response will not be recorded on the input
history stack.
The result is a String, which is whatever the user actually
typed on the command-line in response to the issued prompt.
NOTE: Command-line completion is not supported.

Parameters:
- {prompt} (string)
- {text} (string?)

Return:
(string)

insert({object}, {item} [, {idx}]) insert()

Lua: Prefer table.insert() for lists; blob insertion and
negative indexes differ.

When {object} is a List or a Blob insert {item} at the start
of it.

If {idx} is specified insert {item} before the item with index
{idx}. If {idx} is zero it goes before the first item, just
like omitting {idx}. A negative {idx} is also possible, see
list-index. -1 inserts just before the last item.

Returns the resulting List or Blob. Examples:

```
let mylist = insert([2, 3, 5], 1)
call insert(mylist, 4, -1)
call insert(mylist, 6, len(mylist))
```

The last example can be done simpler with add().
Note that when {item} is a List it is inserted as a single
item. Use extend() to concatenate Lists.

Parameters:
- {object} (any)

- {item} (any)

- {idx} (integer?)

Return:
(any)

interrupt() interrupt()

Interrupt script execution. It works more or less like the user typing CTRL-C, most commands won't execute and control returns to the user. This is useful to abort execution from lower down, e.g. in an autocommand. Example:

```
function s:check_typoname(file)
   if fnamemodify(a:file, ':t') == '['
       echomsg 'Maybe typo'
       call interrupt()
   endif
endfunction
au BufWritePre * call s:check_typoname(expand('<amatch>'))
```

Return:
(any)

invert({expr}) invert()

Lua: Prefer bit.bnot().

Bitwise invert. The argument is converted to a number. A List, Dict or Float argument causes an error. Example:

```
let bits = invert(bits)
```

Parameters:
- {expr} (integer)

Return:
(integer)

isabsolutepath({path}) isabsolutepath()

Checks whether {path} is an absolute path. Returns TRUE if so, FALSE otherwise.
On Unix, a path is considered absolute when it starts with '/'.
On MS-Windows, it is considered absolute when it starts with an optional drive prefix and is followed by a '\' or '/'. UNC paths are always absolute.
Example:

```
echo isabsolutepath('/usr/share/')      " 1
echo isabsolutepath('./foobar')         " 0
echo isabsolutepath('C:\Windows')       " 1
echo isabsolutepath('foobar')           " 0
echo isabsolutepath('\\remote\file')    " 1
```

Attributes:
api-fast

Parameters:
- {path} (string)

Return:
(0|1)

isdirectory({directory}) isdirectory()

Lua: Prefer uv.fs_stat() and check type == "directory".

Returns TRUE if {directory} exists, or FALSE if it doesn't exist or isn't a directory. {directory} is any expression, which is used as a String.

Attributes:
api-fast

Parameters:
- {directory} (string)

Return:
(0|1)

isinf({expr}) isinf()

Lua: Prefer
expr == math.huge and 1 or expr == -math.huge and -1 or 0.

Return 1 if {expr} is a positive infinity, or -1 a negative
infinity, otherwise 0.

```
echo isinf(1.0 / 0.0)
```

```
1
```

```
echo isinf(-1.0 / 0.0)
```

```
-1
```

Parameters:
- {expr} (number)

Return:
(1|0|-1)

islocked({expr}) islocked() E786

Returns TRUE if {expr} is the name of a locked variable,
else FALSE.
The string argument {expr} must be the name of a variable,
List item or Dictionary entry, not the variable itself!
Example:

```
let alist = [0, ['a', 'b'], 2, 3]
lockvar 1 alist
echo islocked('alist')          " 1
echo islocked('alist[1]')       " 0
```

When {expr} is a variable that does not exist you get an error
message. Use exists() to check for existence.

Parameters:
- {expr} (any)

Return:
(0|1)

isnan({expr}) isnan()

Lua: Prefer expr ~= expr.

Return TRUE if {expr} is a float with value NaN.

```
echo isnan(0.0 / 0.0)
```

```
1
```

Parameters:
- {expr} (number)

Return:
(0|1)

items({expr}) items()

Lua: Prefer pairs() or ipairs().

Return a List with all key/index and value pairs of {expr}.
Each List item is a list with two items:
- for a Dict: the key and the value

- for a List, Blob or String: the index and the value

The returned List is in arbitrary order for a Dict,
otherwise it's in ascending order of the index.

Also see keys() and values().

Example:

```
let mydict = #{a: 'red', b: 'blue'}
for [key, value] in items(mydict)
   echo $"{key} = {value}"
endfor
echo items([1, 2, 3])
echo items("foobar")
echo items(0z0102)
```

Parameters:
- {expr} (table|string)

Return:
(any)

jobpid({job}) jobpid()

Lua: Prefer vim.system() object .pid; job-id and channel
semantics differ.

Return the PID (process id) of job-id {job}.

Parameters:

- {job} (integer)

Return:

(integer)

jobresize({job}, {width}, {height}) jobresize()

Resize the pseudo terminal window of job-id {job} to {width}
columns and {height} rows.
Fails if the job was not started with "pty":v:true.

Parameters:

- {job} (integer)
- {width} (integer)
- {height} (integer)

Return:

(any)

jobstart({cmd} [, {opts}]) jobstart()

Lua: Prefer vim.system() for simple processes; RPC,
PTY/term, streaming callbacks, and shell-string behavior
differ.

Spawns {cmd} as a job.
If {cmd} is a List it runs directly (no 'shell').
If {cmd} is a String it runs in the 'shell', like this:

```
call jobstart(split(&shell) + split(&shellcmdflag) + ['{cmd}'])
```

(See shell-unquoting for details.)

Example: start a job and handle its output:

```
call jobstart(['nvim', '-h'], {'on_stdout':{j,d,e->append(line('.'),d)}})
```

Example: start a job in a terminal connected to the current buffer:

```
call jobstart(['nvim', '-h'], {'term':v:true})
```

Returns job-id on success, 0 on invalid arguments (or job
table is full), -1 if {cmd}[0] or 'shell' is not executable.
The returned job-id is a valid channel-id representing the
job's stdio streams. Use chansend() (or rpcnotify() and
rpcrequest() if "rpc" was enabled) to send data to stdin and
chanclose() to close the streams without stopping the job.

See job-control and RPC.

NOTE: on Windows if {cmd} is a List:

- cmd[0] must be an executable (not a "built-in"). If it is

  in $PATH it can be called by name, without an extension:

  ```
  call jobstart(['ping', 'neovim.io'])
  ```

  If it is a full or partial path, extension is required:

  ```
  call jobstart(['System32\ping.exe', 'neovim.io'])
  ```

- {cmd} is collapsed to a string of quoted args as expected

  by CommandLineToArgvW https://msdn.microsoft.com/bb776391
  unless cmd[0] is some form of "cmd.exe".

jobstart-env

The job environment is initialized as follows:

$NVIM	is set to v:servername of the parent Nvim
$NVIM_LISTEN_ADDRESS	is unset
$NVIM_LOG_FILE	is unset
$VIM	is unset
$VIMRUNTIME	is unset

You can set these with the env option.

jobstart-options

{opts} is a dictionary with these keys:

clear_env: (boolean) env defines the job environment exactly, instead of merging current environment.
cwd: (string, default=|current-directory|) Working directory of the job.
detach: (boolean) Detach the job process: it will not be killed when Nvim exits. If the process exits before Nvim, on_exit will be invoked.
env: (dict) Map of environment variable name:value pairs extending (or replace with "clear_env") the current environment. jobstart-env
height: (number) Height of the pty terminal.
on_exit: (function) Callback invoked when the job exits.
on_stdout: (function) Callback invoked when the job emits stdout data.
on_stderr: (function) Callback invoked when the job emits stderr data.
overlapped: (boolean) Sets FILE_FLAG_OVERLAPPED for the stdio passed to the child process. Only on MS-Windows; ignored on other platforms.
pty: (boolean) Connect the job to a new pseudo terminal, and its streams to the master file descriptor. on_stdout receives all output, on_stderr is ignored. **Note:** if the child writes a query (DA1, OSC, …), it may hang or timeout waiting for a response! To avoid that, on_stdout should reply via nvim_chan_send() on the child's stdin. See terminal-start terminal-concepts
rpc: (boolean) Use msgpack-rpc to communicate with the job over stdio. Then on_stdout is ignored, but on_stderr can still be used.
stderr_buffered: (boolean) Collect data until EOF (stream closed) before invoking on_stderr. channel-buffered
stdout_buffered: (boolean) Collect data until EOF (stream closed) before invoking on_stdout. channel-buffered
stdin: (string) Either "pipe" (default) to connect the job's stdin to a channel or "null" to disconnect stdin.
term: (boolean) Spawns {cmd} in a new pseudo-terminal session connected to the current (unmodified) buffer. Implies "pty". Defaults "height" and "width" to the current window dimensions. Defaults $TERM to "xterm-256color".
width: (number) Width of the pty pseudo-terminal.

{opts} is passed as self dictionary to the callback; the caller may set other keys to pass application-specific data.

Returns:

- channel-id on success
- 0 on invalid arguments
- -1 if {cmd}[0] is not executable.

See also job-control, channel, msgpack-rpc.

Parameters:

- {cmd} (string|string[])
- {opts} (table?)

Return:

(integer)

jobstop({id}) jobstop()

Lua: Prefer SystemObj:kill() on a vim.system() object; job-id semantics differ.

Stop job-id {id} by sending SIGTERM to the job process. If the process does not terminate after a timeout then SIGKILL will be sent. When the job terminates its on_exit handler (if any) will be invoked.
See job-control.

Returns 1 for valid job id, 0 for invalid id, including jobs have exited or stopped.

Parameters:

- {id} (integer)

Return:

(integer)

jobwait({jobs} [, {timeout}]) jobwait()

Lua: Prefer SystemObj:wait() on a vim.system() object;
list input and status values differ.

Waits for jobs and their on_exit handlers to complete.

{jobs} is a List of job-ids to wait for.
{timeout} is the maximum waiting time in milliseconds. If
omitted or -1, wait forever.

Timeout of 0 can be used to check the status of a job:

```
let running = jobwait([{job-id}], 0)[0] == -1
```

During jobwait() callbacks for jobs not in the {jobs} list may
be invoked. The screen will not redraw unless :redraw is
invoked by a callback.

Returns a list of len({jobs}) integers, where each integer is
the status of the corresponding job:
        Exit-code, if the job exited
        -1 if the timeout was exceeded
        -2 if the job was interrupted (by CTRL-C)
        -3 if the job-id is invalid

Parameters:
- {jobs} (integer[])

- {timeout} (integer?)

Return:
  (integer[])

join({list} [, {sep}]) join()

Lua: Prefer table.concat() for string lists; non-string
values stringify differently.

Join the items in {list} together into one String.
When {sep} is specified it is put in between the items. If
{sep} is omitted a single space is used.
Note that {sep} is not added at the end. You might want to
add it there too:

```
let lines = join(mylist, "\n") .. "\n"
```

String items are used as-is. Lists and Dictionaries are
converted into a string like with string().
The opposite function is split().

Parameters:
- {list} (any[])

- {sep} (string?)

Return:
  (string)

json_decode({expr}) json_decode()

Lua: Prefer vim.json.decode().

Convert {expr} from JSON object. Accepts readfile()-style
list as the input, as well as regular string. May output any
Vim value. In the following cases it will output
msgpack-special-dict:
1. Dictionary contains duplicate key.

2. String contains NUL byte. Two special dictionaries: for

    dictionary and for string will be emitted in case string
    with NUL byte was a dictionary key.

Note: function treats its input as UTF-8 always. The JSON
standard allows only a few encodings, of which UTF-8 is
recommended and the only one required to be supported.
Non-UTF-8 characters are an error.

Parameters:
- {expr} (any)

Return:
  (any)

json_encode({expr}) json_encode()

Lua: Prefer vim.json.encode().

Convert {expr} into a JSON string. Accepts
msgpack-special-dict as the input. Will not convert

Funcrefs, mappings with non-string keys (can be created as msgpack-special-dict), values with self-referencing containers, strings which contain non-UTF-8 characters, pseudo-UTF-8 strings which contain codepoints reserved for surrogate pairs (such strings are not valid UTF-8 strings). Non-printable characters are converted into "\u1234" escapes or special escapes like "\t", other are dumped as-is. Blobs are converted to arrays of the individual bytes.

Parameters:
- {expr} (any)

Return:
(string)

keys({dict}) keys()

Lua: Prefer vim.tbl_keys().

Return a List with all the keys of {dict}. The List is in arbitrary order. Also see items() and values().

Parameters:
- {dict} (table)

Return:
(string[])

keytrans({string}) keytrans()

Turn the internal byte representation of keys into a form that can be used for :map. E.g.

```
let xx = "\<C-Home>"
echo keytrans(xx)
```

<C-Home>

Attributes:
api-fast

Parameters:
- {string} (string)

Return:
(string)

len({expr}) len() E701

Lua: Prefer Lua # operator for strings and lists or #vim.tbl_keys(expr) for dicts.

Returns the length of the argument.
When {expr} is a String or a Number the length in bytes is used, as with strlen().
When {expr} is a List the number of items in the List is returned.
When {expr} is a Blob the number of bytes is returned.
When {expr} is a Dictionary the number of entries in the Dictionary is returned.
Otherwise an error is given and returns zero.

Parameters:
- {expr} (any[])

Return:
(integer)

libcall({libname}, {funcname}, {argument}) libcall() E364 E368

Lua: Prefer package.loadlib() or ffi.load().

Call function {funcname} in the run-time library {libname} with single argument {argument}.
This is useful to call functions in a library that you especially made to be used with Vim. Since only one argument is possible, calling standard library functions is rather limited.
The result is the String returned by the function. If the function returns NULL, this will appear as an empty string "" to Vim.
If the function returns a number, use libcallnr()!
If {argument} is a number, it is passed to the function as an int; if {argument} is a string, it is passed as a null-terminated string.

libcall() allows you to write your own 'plug-in' extensions to Vim without having to recompile the program. It is NOT a means to call system functions! If you try to do so Vim will very probably crash.

For Win32, the functions you write must be placed in a DLL and use the normal C calling convention (NOT Pascal which is used in Windows System DLLs). The function must take exactly

one parameter, either a character pointer or a long integer, and must return a character pointer or NULL. The character pointer returned must point to memory that will remain valid after the function has returned (e.g. in static data in the DLL). If it points to allocated memory, that memory will leak away. Using a static buffer in the function should work, it's then freed when the DLL is unloaded.

WARNING: If the function returns a non-valid pointer, Vim may crash! This also happens if the function returns a number, because Vim thinks it's a pointer.
For Win32 systems, {libname} should be the filename of the DLL without the ".DLL" suffix. A full path is only required if the DLL is not in the usual places.
For Unix: When compiling your own plugins, remember that the object code must be compiled as position-independent ('PIC').
Examples:

```
echo libcall("libc.so", "getenv", "HOME")
```

Parameters:
- {libname} (string)
- {funcname} (string)
- {argument} (any)

Return:
(any)

libcallnr({libname}, {funcname}, {argument}) libcallnr()

Lua: Prefer package.loadlib() or ffi.load().

Just like libcall(), but used for a function that returns an int instead of a string.
Examples:

```
echo libcallnr("/usr/lib/libc.so", "getpid", "")
call libcallnr("libc.so", "printf", "Hello World!\n")
call libcallnr("libc.so", "sleep", 10)
```

Parameters:
- {libname} (string)
- {funcname} (string)
- {argument} (any)

Return:
(any)

line({expr} [, {winid}]) line()

Lua: Prefer nvim_win_get_cursor() for "." or nvim_buf_line_count() for "$"; marks, lists, and other position expressions differ.

See getpos() for accepted positions.

To get the column number use col(). To get both use getpos().

With the optional {winid} argument the values are obtained for that window instead of the current window.

Returns 0 for invalid values of {expr} and {winid}.

Examples:

```
echo line(".")                  " line number of the cursor
echo line(".", winid)           " idem, in window "winid"
echo line("'t")                 " line number of mark t
echo line("'" .. marker)        " line number of mark marker
```

To jump to the last known position when opening a file see last-position-jump.

Parameters:
- {expr} (string|integer[])
- {winid} (integer?)

Return:
(integer)

line2byte({lnum}) line2byte()

Lua: Prefer nvim_buf_get_offset() with lnum - 1;
'fileformat' and invalid-line semantics differ.

Return the byte count from the start of the buffer for line
{lnum}. This includes the end-of-line character, depending on
the 'fileformat' option for the current buffer. The first
line returns 1. UTF-8 encoding is used, 'fileencoding' is
ignored. This can also be used to get the byte count for the
line just below the last line:

```
echo line2byte(line("$") + 1)
```

This is the buffer size plus one. If 'fileencoding' is empty
it is the file size plus one. {lnum} is used like with
getline(). When {lnum} is invalid -1 is returned.
Also see byte2line(), go and :goto.

Parameters:
- {lnum} (integer|string)

Return:
(integer)

lispindent({lnum}) lispindent()

Get the amount of indent for line {lnum} according the lisp
indenting rules, as with 'lisp'.
The indent is counted in spaces, the value of 'tabstop' is
relevant. {lnum} is used just like in getline().
When {lnum} is invalid, -1 is returned.

Parameters:
- {lnum} (integer|string)

Return:
(integer)

list2blob({list}) list2blob()

Return a Blob concatenating all the number values in {list}.
Examples:

```
echo list2blob([1, 2, 3, 4])    " returns 0z01020304
echo list2blob([])              " returns 0z
```

Returns an empty Blob on error. If one of the numbers is
negative or more than 255 error E1239 is given.

blob2list() does the opposite.

Parameters:
- {list} (any[])

Return:
(string)

list2str({list} [, {utf8}]) list2str()

Converts each codepoint in {list} to a UTF-8 character and
returns the concatenated string. Examples:

```
echo list2str([32])             " returns " "
echo list2str([65, 66, 67])     " returns "ABC"
```

The same can be done (slowly) with:

```
echo join(map(list, {nr, val -> nr2char(val)}), '')
```

str2list() does the opposite.

UTF-8 encoding is always used, {utf8} option has no effect,
and exists only for backwards-compatibility.
With UTF-8 composing characters work as expected:

```
echo list2str([97, 769])        " returns "á"
```

Returns an empty string on error.

Parameters:
- {list} (any[])
- {utf8} (boolean?)

Return:
(string)

localtime() localtime()

Lua: Prefer os.time().

Return the current time, measured as seconds since 1st Jan 1970. See also strftime(), strptime() and getftime().

Return:
(integer)

log({expr}) log()

Lua: Prefer math.log().

Return the natural logarithm (base e) of {expr} as a Float. {expr} must evaluate to a Float or a Number in the range (0, inf].
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo log(10)
```

2.302585

```
echo log(exp(5))
```

5.0

Parameters:
- {expr} (number)

Return:
(number)

log10({expr}) log10()

Lua: Prefer math.log10().

Return the logarithm of Float {expr} to base 10 as a Float. {expr} must evaluate to a Float or a Number.
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo log10(1000)
```

3.0

```
echo log10(0.01)
```

-2.0

Parameters:
- {expr} (number)

Return:
(number)

luaeval({expr} [, {expr}]) luaeval()

Lua: Prefer plain Lua expressions and values directly.

Evaluate Lua expression {expr} and return its result converted to Vim data structures. See lua-eval for details.

See also v:lua-call.

Parameters:
- {expr} (string)
- {expr1} (any[]?)

Return:
(any)

map({expr1}, {expr2}) map()

Lua: Prefer vim.tbl_map() for array-like tables; dict, string, blob, and in-place behavior differ.

{expr1} must be a List, String, Blob or Dictionary.
When {expr1} is a List or Dictionary, replace each item in {expr1} with the result of evaluating {expr2}.
For a Blob each byte is replaced.
For a String, each character, including composing characters, is replaced.
If the item type changes you may want to use mapnew() to create a new List or Dictionary.

{expr2} must be a String or Funcref.

If {expr2} is a String, inside {expr2} v:val has the value of the current item. For a Dictionary v:key has the key of the current item and for a List v:key has the index of the current item. For a Blob v:key has the index of the

current byte. For a String v:key has the index of the
current character.
Example:

```
call map(mylist, '"> " .. v:val .. " <"')
```

This puts "> " before and " <" after each item in "mylist".

Note that {expr2} is the result of an expression and is then
used as an expression again. Often it is good to use a
literal-string to avoid having to double backslashes. You
still have to double single (') quotes, though.

If {expr2} is a Funcref it is called with two arguments:
1. The key or the index of the current item.

2. the value of the current item.

The function must return the new value of the item. Example
that changes each value by "key-value":

```
func KeyValue(key, val)
  return a:key .. '-' .. a:val
endfunc
call map(myDict, function('KeyValue'))
```

It is shorter when using a lambda:

```
call map(myDict, {key, val -> key .. '-' .. val})
```

If you do not use "val" you can leave it out:

```
call map(myDict, {key -> 'item: ' .. key})
```

If you do not use "key" you can use a short name:

```
call map(myDict, {_, val -> 'item: ' .. val})
```

The operation is done in-place for a List and Dictionary.
If you want it to remain unmodified make a copy first:

```
let tlist = map(copy(mylist), ' v:val .. "\t"')
```

Returns {expr1}, the List or Dictionary that was filtered,
or a new Blob or String.
When an error is encountered while evaluating {expr2} no
further items in {expr1} are processed.
When {expr2} is a Funcref errors inside a function are
ignored, unless it was defined with the "abort" flag.

Parameters:

- {expr1} (string|table|any[])
- {expr2} (string|function)

Return:

(any)

maparg({name} [, {mode} [, {abbr} [, {dict}]]]) maparg()

When {dict} is omitted or zero: Return the rhs of mapping
{name} in mode {mode}. The returned String has special
characters translated like in the output of the ":map" command
listing. When {dict} is TRUE a dictionary is returned, see
below. To get a list of all mappings see maplist().

When there is no mapping for {name}, an empty String is
returned if {dict} is FALSE, otherwise returns an empty Dict.
When the mapping for {name} is empty, then "<Nop>" is
returned.

The {name} can have special key names, like in the ":map"
command.

{mode} can be one of these strings:

"n"	Normal
"v"	Visual (including Select)
"o"	Operator-pending
"i"	Insert
"c"	Cmd-line
"s"	Select
"x"	Visual
"l"	langmap language-mapping
"t"	Terminal
""	Normal, Visual and Operator-pending

When {mode} is omitted, the modes for "" are used.

When {abbr} is there and it is TRUE use abbreviations
instead of mappings.

When {dict} is TRUE, return a dictionary describing the
mapping, with these items: mapping-dict

```
  "lhs"      The {lhs} of the mapping as it would be typed
  "lhsraw"   The {lhs} of the mapping as raw bytes
  "lhsrawalt" The {lhs} of the mapping as raw bytes, alternate
              form, only present when it differs from "lhsraw"
  "rhs"      The {rhs} of the mapping as typed.
  "callback" Lua function, if RHS was defined as such.
  "silent"   1 for a :map-silent mapping, else 0.
  "noremap"  1 if the {rhs} of the mapping is not remappable.
  "script"   1 if mapping was defined with <script>.
  "expr"     1 for an expression mapping (:map-<expr>).
  "buffer"   1 for a buffer local mapping (:map-local).
  "mode"     Modes for which the mapping is defined.  In
             addition to the modes mentioned above, these
             characters will be used:
             " "     Normal, Visual and Operator-pending
             "!"     Insert and Commandline mode
                     (mapmode-ic)
  "sid"      The script local ID, used for <sid> mappings
             (<SID>).  Negative for special contexts.
  "scriptversion"  The version of the script, always 1.
  "lnum"     The line number in "sid", zero if unknown.
  "nowait"   Do not wait for other, longer mappings.
             (:map-<nowait>).
  "abbr"     True if this is an abbreviation.
  "mode_bits" Nvim's internal binary representation of "mode".
             mapset() ignores this; only "mode" is used.
             See maplist() for usage examples.  The values
             are from src/nvim/state_defs.h and may change in
             the future.
```

The dictionary can be used to restore a mapping with
mapset().

The mappings local to the current buffer are checked first,
then the global mappings.
This function can be used to map a key even when it's already
mapped, and have it do the original mapping too. Sketch:

```
exe 'nnoremap <Tab> ==' .. maparg('<Tab>', 'n')
```

Parameters:

- {name} (string)
- {mode} (string?)
- {abbr} (boolean?)
- {dict} (false?)

Return:

(string)

mapcheck({name} [, {mode} [, {abbr}]]) mapcheck()

Check if there is a mapping that matches with {name} in mode
{mode}. See maparg() for {mode} and special names in
{name}.
When {abbr} is there and it is non-zero use abbreviations
instead of mappings.
A match happens with a mapping that starts with {name} and
with a mapping which is equal to the start of {name}.

matches mapping "a"	"ab"	"abc"	
mapcheck("a")	yes	yes	yes
mapcheck("abc")	yes	yes	yes
mapcheck("ax")	yes	no	no
mapcheck("b")	no	no	no

The difference with maparg() is that mapcheck() finds a
mapping that matches with {name}, while maparg() only finds a
mapping for {name} exactly.
When there is no mapping that starts with {name}, an empty
String is returned. If there is one, the RHS of that mapping
is returned. If there are several mappings that start with
{name}, the RHS of one of them is returned. This will be
"<Nop>" if the RHS is empty.
The mappings local to the current buffer are checked first,
then the global mappings.
This function can be used to check if a mapping can be added
without being ambiguous. Example:

```
if mapcheck("_vv") == ""
   map _vv :set guifont=7x13<CR>
endif
```

This avoids adding the "_vv" mapping when there already is a mapping for "_v" or for "_vvv".

Parameters:
- {name} (string)
- {mode} (string?)
- {abbr} (boolean?)

Return:
(any)

maplist([{abbr}]) maplist()

Returns a List of all mappings. Each List item is a Dict, the same as what is returned by maparg(), see mapping-dict. When {abbr} is there and it is TRUE use abbreviations instead of mappings.

Example to show all mappings with "MultiMatch" in rhs:

```
echo maplist()->filter({_, m ->
        \ match(get(m, 'rhs', ''), 'MultiMatch') >= 0
        \ })
```

It can be tricky to find mappings for particular :map-modes. mapping-dict's "mode_bits" can simplify this. For example, the mode_bits for Normal, Insert or Command-line modes are 0x19. To find all the mappings available in those modes you can do:

```
let saved_maps = []
for m in maplist()
    if and(m.mode_bits, 0x19) != 0
        eval saved_maps->add(m)
    endif
endfor
echo saved_maps->mapnew({_, m -> m.lhs})
```

The values of the mode_bits are defined in Nvim's src/nvim/state_defs.h file and they can be discovered at runtime using :map-commands and "maplist()". Example:

```
omap xyzzy <Nop>
let op_bit = maplist()->filter(
    \ {_, m -> m.lhs == 'xyzzy'})[0].mode_bits
ounmap xyzzy
echo printf("Operator-pending mode bit: 0x%x", op_bit)
```

Parameters:
- {abbr} (0|1?)

Return:
(table[])

mapnew({expr1}, {expr2}) mapnew()

Lua: Prefer vim.tbl_map() for tables; string, blob, and v:key callback behavior differ.

Like map() but instead of replacing items in {expr1} a new List or Dictionary is created and returned. {expr1} remains unchanged. Items can still be changed by {expr2}, if you don't want that use deepcopy() first.

Parameters:
- {expr1} (any)
- {expr2} (any)

Return:
(any)

mapset({mode}, {abbr}, {dict}) mapset()

mapset({dict})

Restore a mapping from a dictionary, possibly returned by maparg() or maplist(). A buffer mapping, when dict.buffer is true, is set on the current buffer; it is up to the caller to ensure that the intended buffer is the current buffer. This feature allows copying mappings from one buffer to

another.
The dict.mode value may restore a single mapping that covers
more than one mode, like with mode values of '!', ' ', "nox",
or 'v'. E1276

In the first form, {mode} and {abbr} should be the same as
for the call to maparg(). E460
{mode} is used to define the mode in which the mapping is set,
not the "mode" entry in {dict}.
Example for saving and restoring a mapping:

```
let save_map = maparg('K', 'n', 0, 1)
nnoremap K somethingelse
" ...
call mapset('n', 0, save_map)
```

Note that if you are going to replace a map in several modes,
e.g. with :map!, you need to save/restore the mapping for
all of them, when they might differ.

In the second form, with {dict} as the only argument, mode
and abbr are taken from the dict.
Example:

```
let save_maps = maplist()->filter(
                      \ {_, m -> m.lhs == 'K'})
nnoremap K somethingelse
cnoremap K somethingelse2
" ...
unmap K
for d in save_maps
    call mapset(d)
endfor
```

Parameters:
- {dict} (table<string,any>)

Return:
(any)

match({expr}, {pat} [, {start} [, {count}]]) match()

When {expr} is a List then this returns the index of the
first item where {pat} matches. Each item is used as a
String, Lists and Dictionaries are used as echoed.

Otherwise, {expr} is used as a String. The result is a
Number, which gives the index (byte offset) in {expr} where
{pat} matches.

A match at the first character or List item returns zero.
If there is no match -1 is returned.

For getting submatches see matchlist().
Example:

```
echo match("testing", "ing")    " results in 4
echo match([1, 'x'], '\a')      " results in 1
```

See string-match for how {pat} is used.

strpbrk()

Vim doesn't have a strpbrk() function. But you can do:

```
let sepidx = match(line, '[.,;: \t]')
```

strcasestr()

Vim doesn't have a strcasestr() function. But you can add
"\c" to the pattern to ignore case:

```
let idx = match(haystack, '\cneedle')
```

If {start} is given, the search starts from byte index
{start} in a String or item {start} in a List.
The result, however, is still the index counted from the
first character/item. Example:

```
echo match("testing", "ing", 2)
```

result is again "4".

```
echo match("testing", "ing", 4)
```

result is again "4".

```
echo match("testing", "t", 2)
```

result is "3".
For a String, if {start} > 0 then it is like the string starts
{start} bytes later, thus "^" will match at {start}. Except
when {count} is given, then it's like matches before the

{start} byte are ignored (this is a bit complicated to keep it
backwards compatible).
For a String, if {start} < 0, it will be set to 0. For a list
the index is counted from the end.
If {start} is out of range ({start} > strlen({expr}) for a
String or {start} > len({expr}) for a List) -1 is returned.

When {count} is given use the {count}th match. When a match
is found in a String the search for the next one starts one
character further. Thus this example results in 1:

```
echo match("testing", "..", 0, 2)
```

In a List the search continues in the next item.
Note that when {count} is added the way {start} works changes,
see above.

match-pattern

See pattern for the patterns that are accepted.
The 'ignorecase' option is used to set the ignore-caseness of
the pattern. 'smartcase' is NOT used. The matching is always
done like 'magic' is set and 'cpoptions' is empty.
Note that a match at the start is preferred, thus when the
pattern is using "*" (any number of matches) it tends to find
zero matches at the start instead of a number of matches
further down in the text.

Parameters:

- {expr} (string|any[])
- {pat} (string)
- {start} (integer?)
- {count} (integer?)

Return:

(integer)

matchadd() E798 E799 E801 E957

matchadd({group}, {pattern} [, {priority} [, {id} [, {dict}]]])
Defines a pattern to be highlighted in the current window (a
"match"). It will be highlighted with {group}. Returns an
identification number (ID), which can be used to delete the
match using matchdelete(). The ID is bound to the window.
Matching is case sensitive and magic, unless case sensitivity
or magicness are explicitly overridden in {pattern}. The
'magic', 'smartcase' and 'ignorecase' options are not used.
The "Conceal" value is special, it causes the match to be
concealed.

The optional {priority} argument assigns a priority to the
match. A match with a high priority will have its
highlighting overrule that of a match with a lower priority.
A priority is specified as an integer (negative numbers are no
exception). If the {priority} argument is not specified, the
default priority is 10. The priority of 'hlsearch' is zero,
hence all matches with a priority greater than zero will
overrule it. Syntax highlighting (see 'syntax') is a separate
mechanism, and regardless of the chosen priority a match will
always overrule syntax highlighting.

The optional {id} argument allows the request for a specific
match ID. If a specified ID is already taken, an error
message will appear and the match will not be added. An ID
is specified as a positive integer (zero excluded). IDs 1, 2
and 3 are reserved for :match, :2match and :3match,
respectively. 3 is reserved for use by the matchparen
plugin.
If the {id} argument is not specified or -1, matchadd()
automatically chooses a free ID, which is at least 1000.

The optional {dict} argument allows for further custom
values. Currently this is used to specify a match specific
conceal character that will be shown for hl-Conceal
highlighted matches. The dict can have the following members:

conceal	Special character to show instead of the match (only for hl-Conceal highlighted matches, see :syn-cchar)
window	Use this window-number or window-ID instead of the current window.

The number of matches is not limited, as it is the case with
the :match commands.

Returns -1 on error.

Example:

```
highlight MyGroup ctermbg=green guibg=green
let m = matchadd("MyGroup", "TODO")
```

Deletion of the pattern:

```
call matchdelete(m)
```

A list of matches defined by matchadd() and :match are available from getmatches(). All matches can be deleted in one operation by clearmatches().

Parameters:

- {group} (integer|string)
- {pattern} (string)
- {priority} (integer?)
- {id} (integer?)
- {dict} (table?)

Return:

(integer)

matchaddpos({group}, {pos} [, {priority} [, {id} [, {dict}]]]) matchaddpos()

Same as matchadd(), but requires a list of positions {pos} instead of a pattern. This command is faster than matchadd() because it does not handle regular expressions and it sets buffer line boundaries to redraw screen. It is supposed to be used when fast match additions and deletions are required, for example to highlight matching parentheses.

E5030 E5031

{pos} is a list of positions. Each position can be one of these:

- A number. This whole line will be highlighted. The first line has number 1.
- A list with one number, e.g., [23]. The whole line with this number will be highlighted.
- A list with two numbers, e.g., [23, 11]. The first number is the line number, the second one is the column number (first column is 1, the value must correspond to the byte index as col() would return). The character at this position will be highlighted.
- A list with three numbers, e.g., [23, 11, 3]. As above, but the third number gives the length of the highlight in bytes.

Entries with zero and negative line numbers are silently ignored, as well as entries with negative column numbers and lengths.

Returns -1 on error.

Example:

```
highlight MyGroup ctermbg=green guibg=green
let m = matchaddpos("MyGroup", [[23, 24], 34])
```

Deletion of the pattern:

```
call matchdelete(m)
```

Matches added by matchaddpos() are returned by getmatches().

Parameters:

- {group} (integer|string)
- {pos} (any[])
- {priority} (integer?)
- {id} (integer?)

- {dict} (table?)

Return:
(integer|table)

matcharg({nr}) matcharg()

Selects the {nr} match item, as set with a :match,
:2match or :3match command.
Return a List with two elements:
The name of the highlight group used
The pattern used.
When {nr} is not 1, 2 or 3 returns an empty List.
When there is no match item set returns ['', ''].
This is useful to save and restore a :match.
Highlighting matches using the :match commands are limited
to three matches. matchadd() does not have this limitation.

Parameters:
- {nr} (integer)

Return:
(string[])

matchbufline({buf}, {pat}, {lnum}, {end}, [, {dict}]) matchbufline()

Returns the List of matches in lines from {lnum} to {end} in
buffer {buf} where {pat} matches.

{lnum} and {end} can either be a line number or the string "$"
to refer to the last line in {buf}.

The {dict} argument supports following items:
submatches include submatch information (/\()

For each match, a Dict with the following items is returned:
byteidx starting byte index of the match
lnum line number where there is a match
text matched string
Note that there can be multiple matches in a single line.

This function works only for loaded buffers. First call
bufload() if needed.

See match-pattern for information about the effect of some
option settings on the pattern.

When {buf} is not a valid buffer, the buffer is not loaded or
{lnum} or {end} is not valid then an error is given and an
empty List is returned.

Examples:

```
" Assuming line 3 in buffer 5 contains "a"
echo matchbufline(5, '\<\k\+\>', 3, 3)
```

[{'lnum': 3, 'byteidx': 0, 'text': 'a'}]

```
" Assuming line 4 in buffer 10 contains "tik tok"
echo matchbufline(10, '\<\k\+\>', 1, 4)
```

[{'lnum': 4, 'byteidx': 0, 'text': 'tik'}, {'lnum': 4, 'byteidx': 4, 'text': 'tok'}]

If {submatch} is present and is v:true, then submatches like
"\1", "\2", etc. are also returned. Example:

```
" Assuming line 2 in buffer 2 contains "acd"
echo matchbufline(2, '\(a\)\?\(b\)\?\(c\)\?\(.*\)', 2, 2
                        \ {'submatches': v:true})
```

[{'lnum': 2, 'byteidx': 0, 'text': 'acd', 'submatches': ['a', '', 'c', 'd', '', '', '', '', '']}]
The "submatches" List always contains 9 items. If a submatch
is not found, then an empty string is returned for that
submatch.

Parameters:
- {buf} (string|integer)
- {pat} (string)
- {lnum} (string|integer)
- {end} (string|integer)
- {dict} (table?)

Return:
(string[])

matchdelete({id} [, {win}]) matchdelete() E802 E803

Deletes a match with ID {id} previously defined by matchadd() or one of the :match commands. Returns 0 if successful, otherwise -1. See example for matchadd(). All matches can be deleted in one operation by clearmatches(). {win} (if given) is a window-number or window-ID.

Parameters:

- {id} (integer)
- {win} (integer?)

Return:

(any)

matchend({expr}, {pat} [, {start} [, {count}]]) matchend()

Same as match(), but return the index of first character after the match. Example:

```
echo matchend("testing", "ing")
```

results in "7".

strspn() strcspn()

Vim doesn't have a strspn() or strcspn() function, but you can do it with matchend():

```
let span = matchend(line, '[a-zA-Z]')
let span = matchend(line, '[^a-zA-Z]')
```

Except that -1 is returned when there are no matches.

The {start}, if given, has the same meaning as for match().

```
echo matchend("testing", "ing", 2)
```

results in "7".

```
echo matchend("testing", "ing", 5)
```

result is "-1".
When {expr} is a List the result is equal to match().

Parameters:

- {expr} (any)
- {pat} (string)
- {start} (integer?)
- {count} (integer?)

Return:

(integer)

matchfuzzy({list}, {str} [, {dict}]) matchfuzzy()

If {list} is a list of strings, then returns a List with all the strings in {list} that fuzzy match {str}. The strings in the returned list are sorted based on the matching score.

The optional {dict} argument always supports the following items:

matchseq	When this item is present return only matches that contain the characters in {str} in the given sequence.
limit	Maximum number of matches in {list} to be returned. Zero means no limit.

If {list} is a list of dictionaries, then the optional {dict} argument supports the following additional items:

key	Key of the item which is fuzzy matched against {str}. The value of this item should be a string.
text_cb	Funcref that will be called for every item in {list} to get the text for fuzzy matching. This should accept a dictionary item as the argument and return the text for that item to use for fuzzy matching.

{str} is treated as a literal string and regular expression matching is NOT supported. Only the first 1024 characters of {str} and of each item in {list} are used for matching; characters beyond that are ignored.

When {str} has multiple words each separated by white space, then the list of strings that have all the words is returned.

If there are no matching strings or there is an error, then an empty list is returned.

When {limit} is given, matchfuzzy() will find up to this number of matches in {list} and return them in sorted order.

Refer to fuzzy-matching for more information about fuzzy matching strings.

Example:

```
echo matchfuzzy(["clay", "crow"], "cay")
```

results in ["clay"].

```
echo getbufinfo()->map({_, v -> v.name})->matchfuzzy("ndl")
```

results in a list of buffer names fuzzy matching "ndl".

```
echo getbufinfo()->matchfuzzy("ndl", {'key' : 'name'})
```

results in a list of buffer information dicts with buffer names fuzzy matching "ndl".

```
echo getbufinfo()->matchfuzzy("spl",
                            \ {'text_cb' : {v -> v.name}})
```

results in a list of buffer information dicts with buffer names fuzzy matching "spl".

```
echo v:oldfiles->matchfuzzy("test")
```

results in a list of file names fuzzy matching "test".

```
let l = readfile("buffer.c")->matchfuzzy("str")
```

results in a list of lines in "buffer.c" fuzzy matching "str".

```
echo ['one two', 'two one']->matchfuzzy('two one')
```

results in ['two one', 'one two'] .

```
echo ['one two', 'two one']->matchfuzzy('two one',
                            \ {'matchseq': 1})
```

results in ['two one'].

Parameters:

- {list} (any[])
- {str} (string)
- {dict} (table?)

Return:

(table)

matchfuzzypos({list}, {str} [, {dict}]) matchfuzzypos()

Same as matchfuzzy(), but returns the list of matched strings, the list of character positions where characters in {str} matches and a list of matching scores. You can use byteidx() to convert a character position to a byte position.

If {str} matches multiple times in a string, then only the positions for the best match is returned.

If there are no matching strings or there is an error, then a list with three empty list items is returned.

Example:

```
echo matchfuzzypos(['testing'], 'tsg')
```

results in [["testing"], [[0, 2, 6]], [99]]

```
echo matchfuzzypos(['clay', 'lacy'], 'la')
```

results in [["lacy", "clay"], [[0, 1], [1, 2]], [153, 133]]

```
echo [{'text': 'hello', 'id' : 10}]
        \ ->matchfuzzypos('ll', {'key' : 'text'})
```

results in [[{"id": 10, "text": "hello"}], [[2, 3]], [127]]

Parameters:

- {list} (any[])

- {str} (string)

- {dict} (table?)

Return:
(table)

matchlist({expr}, {pat} [, {start} [, {count}]]) matchlist()

Same as match(), but return a List. The first item in the list is the matched string, same as what matchstr() would return. Following items are submatches, like "\1", "\2", etc. in :substitute. When an optional submatch didn't match an empty string is used. Example:

```
echo matchlist('acd', '\(a\)\?\(b\)\?\(c\)\?\(.*\)')
```

Results in: ['acd', 'a', '', 'c', 'd', '', '', '', '', ''] When there is no match an empty list is returned.

You can pass in a List, but that is not very useful.

Parameters:
- {expr} (any)

- {pat} (string)

- {start} (integer?)

- {count} (integer?)

Return:
(string[])

matchstr({expr}, {pat} [, {start} [, {count}]]) matchstr()

Same as match(), but return the matched string. Example:

```
echo matchstr("testing", "ing")
```

results in "ing". When there is no match "" is returned. The {start}, if given, has the same meaning as for match().

```
echo matchstr("testing", "ing", 2)
```

results in "ing".

```
echo matchstr("testing", "ing", 5)
```

result is "". When {expr} is a List then the matching item is returned. The type isn't changed, it's not necessarily a String.

Parameters:
- {expr} (any)

- {pat} (string)

- {start} (integer?)

- {count} (integer?)

Return:
(string)

matchstrlist({list}, {pat} [, {dict}]) matchstrlist()

Returns the List of matches in {list} where {pat} matches. {list} is a List of strings. {pat} is matched against each string in {list}.

The {dict} argument supports following items:
```
    submatches  include submatch information (/\()
```

For each match, a Dict with the following items is returned:
```
    byteidx     starting byte index of the match.
    idx         index in {list} of the match.
    text        matched string
    submatches  a List of submatches.  Present only if
                "submatches" is set to v:true in {dict}.
```

See match-pattern for information about the effect of some option settings on the pattern.

Example:

```
echo matchstrlist(['tik tok'], '\<\k\+\>')
```

```
    [{'idx': 0, 'byteidx': 0, 'text': 'tik'}, {'idx': 0, 'byteidx': 4, 'text': 'tok'}]
```

```
echo matchstrlist(['a', 'b'], '\<\k\+\>')
```

```
    [{'idx': 0, 'byteidx': 0, 'text': 'a'}, {'idx': 1, 'byteidx': 0, 'text': 'b'}]
```

If "submatches" is present and is v:true, then submatches like
"\1", "\2", etc. are also returned. Example:

```
echo matchstrlist(['acd'], '\(a\)\?\(b\)\?\(c\)\?\(.*\)',
                          \ #{submatches: v:true})
```

```
    [{'idx': 0, 'byteidx': 0, 'text': 'acd', 'submatches': ['a', '', 'c', 'd', '', '', '', '', '']}]
```

The "submatches" List always contains 9 items. If a submatch
is not found, then an empty string is returned for that
submatch.

Parameters:

- {list} (string[])
- {pat} (string)
- {dict} (table?)

Return:

(string[])

matchstrpos({expr}, {pat} [, {start} [, {count}]]) matchstrpos()

Same as matchstr(), but return the matched string, the start
position and the end position of the match. Example:

```
echo matchstrpos("testing", "ing")
```

results in ["ing", 4, 7].
When there is no match ["", -1, -1] is returned.
The {start}, if given, has the same meaning as for match().

```
echo matchstrpos("testing", "ing", 2)
```

results in ["ing", 4, 7].

```
echo matchstrpos("testing", "ing", 5)
```

result is ["", -1, -1].
When {expr} is a List then the matching item, the index
of first item where {pat} matches, the start position and the
end position of the match are returned.

```
echo matchstrpos([1, '__x'], '\a')
```

result is ["x", 1, 2, 3].
The type isn't changed, it's not necessarily a String.

Parameters:

- {expr} (any)
- {pat} (string)
- {start} (integer?)
- {count} (integer?)

Return:

(table)

max({expr}) max()

Lua: Prefer math.max() with unpack(expr) for lists; dict
values and empty inputs need custom handling.

Return the maximum value of all items in {expr}. Example:

```
echo max([apples, pears, oranges])
```

{expr} can be a List or a Dictionary. For a Dictionary,
it returns the maximum of all values in the Dictionary.
If {expr} is neither a List nor a Dictionary, or one of the
items in {expr} cannot be used as a Number this results in
an error. An empty List or Dictionary results in zero.

Parameters:

- {expr} (any)

Return:
(number)

menu_get({path} [, {modes}]) menu_get()

Returns a List of Dictionaries describing menus (defined by :menu, :amenu, …), including hidden-menus.

{path} matches a menu by name, or all menus if {path} is an empty string. Example:

```
echo menu_get('File','')
echo menu_get('')
```

{modes} is a string of zero or more modes (see maparg() or creating-menus for the list of modes). "a" means "all".

Example:

```
nnoremenu &Test.Test inormal
inoremenu Test.Test insert
vnoremenu Test.Test x
echo menu_get("")
```

returns something like this:

```
[ {
  "hidden": 0,
  "name": "Test",
  "priority": 500,
  "shortcut": 84,
  "submenus": [ {
    "hidden": 0,
    "mappings": {
      i": {
        "enabled": 1,
        "noremap": 1,
        "rhs": "insert",
        "sid": 1,
        "silent": 0
      },
      n": { ... },
      s": { ... },
      v": { ... }
    },
    "name": "Test",
    "priority": 500,
    "shortcut": 0
  } ]
} ]
```

Parameters:
- {path} (string)

- {modes} (string?)

Return:
(any)

menu_info({name} [, {mode}]) menu_info()

Return information about the specified menu {name} in mode {mode}. The menu name should be specified without the shortcut character ('&'). If {name} is "", then the top-level menu names are returned.

{mode} can be one of these strings:

"n"	Normal
"v"	Visual (including Select)
"o"	Operator-pending
"i"	Insert
"c"	Cmd-line
"s"	Select
"x"	Visual
"t"	Terminal-Job
""	Normal, Visual and Operator-pending
"!"	Insert and Cmd-line

When {mode} is omitted, the modes for "" are used.

Returns a Dictionary containing the following items:

accel	menu item accelerator text menu-text
display	display name (name without '&')
enabled	v:true if this menu item is enabled Refer to :menu-enable
icon	name of the icon file (for toolbar) toolbar-icon
iconidx	index of a built-in icon

modes modes for which the menu is defined. In addition to the modes mentioned above, these characters will be used:
" " Normal, Visual and Operator-pending
name menu item name.
noremenu v:true if the {rhs} of the menu item is not remappable else v:false.
priority menu order priority menu-priority
rhs right-hand-side of the menu item. The returned string has special characters translated like in the output of the ":menu" command listing. When the {rhs} of a menu item is empty, then "<Nop>" is returned.
script v:true if script-local remapping of {rhs} is allowed else v:false. See :menu-script.
shortcut shortcut key (character after '&' in the menu name) menu-shortcut
silent v:true if the menu item is created with <silent> argument :menu-silent
submenus List containing the names of all the submenus. Present only if the menu item has submenus.

Returns an empty dictionary if the menu item is not found.

Examples:

```
echo menu_info('Edit.Cut')
echo menu_info('File.Save', 'n')

" Display the entire menu hierarchy in a buffer
func ShowMenu(name, pfx)
  let m = menu_info(a:name)
  call append(line('$'), a:pfx .. m.display)
  for child in m->get('submenus', [])
    call ShowMenu(a:name .. '.' .. escape(child, '.'),
                               \ a:pfx .. '    ')
  endfor
endfunc
new
for topmenu in menu_info('').submenus
  call ShowMenu(topmenu, '')
endfor
```

Parameters:
- {name} (string)

- {mode} (string?)

Return:
(any)

min({expr}) min()

Lua: Prefer math.min() with unpack(expr) for lists; dict values and empty inputs need custom handling.

Return the minimum value of all items in {expr}. Example:

```
echo min([apples, pears, oranges])
```

{expr} can be a List or a Dictionary. For a Dictionary, it returns the minimum of all values in the Dictionary. If {expr} is neither a List nor a Dictionary, or one of the items in {expr} cannot be used as a Number this results in an error. An empty List or Dictionary results in zero.

Parameters:
- {expr} (any)

Return:
(number)

mkdir({name} [, {flags} [, {prot}]]) mkdir() E739

Lua: Prefer vim.fs.mkdir(); "D", "R", and return semantics differ.

Create directory {name}.

When {flags} is present it must be a string. An empty string has no effect.

{flags} can contain these character flags:
"p" intermediate directories will be created as necessary
"D" {name} will be deleted at the end of the current function, but not recursively :defer
"R" {name} will be deleted recursively at the end of the current function :defer

Note that when {name} has more than one part and "p" is used some directories may already exist. Only the first one that

is created and what it contains is scheduled to be deleted.
E.g. when using:

```
call mkdir('subdir/tmp/autoload', 'pR')
```

and "subdir" already exists then "subdir/tmp" will be
scheduled for deletion, like with:

```
defer delete('subdir/tmp', 'rf')
```

If {prot} is given it is used to set the protection bits of
the new directory. The default is 0o755 (rwxr-xr-x: r/w for
the user, readable for others). Use 0o700 to make it
unreadable for others. This is used for the newly created
directories. **Note:** umask is applied to {prot} (on Unix).
Example:

```
call mkdir($HOME .. "/tmp/foo/bar", "p", 0o700)
```

This function is not available in the sandbox.

If you try to create an existing directory with {flags} set to
"p" mkdir() will silently exit.

The function result is a Number, which is TRUE if the call was
successful or FALSE if the directory creation failed or partly
failed.

Parameters:

- {name} (string)
- {flags} (string?)
- {prot} (string?)

Return:

(integer)

mode([{expr}]) mode()

Lua: Prefer nvim_get_mode(); use the .mode field. It
always returns the full mode string, unlike mode() without a
non-zero {expr}.

Return a string that indicates the current mode.
If {expr} is supplied and it evaluates to a non-zero Number or
a non-empty String (non-zero-arg), then the full mode is
returned, otherwise only the first letter is returned.
Also see state().

```
n        Normal
no       Operator-pending
nov      Operator-pending (forced charwise o_v)
noV      Operator-pending (forced linewise o_V)
noCTRL-V Operator-pending (forced blockwise o_CTRL-V)
             CTRL-V is one character
niI      Normal using i_CTRL-O in Insert-mode
niR      Normal using i_CTRL-O in Replace-mode
niV      Normal using i_CTRL-O in Virtual-Replace-mode
nt       Normal in terminal-emulator (insert goes to
             Terminal mode)
ntT      Normal using t_CTRL-\_CTRL-O in Terminal-mode
v        Visual by character
vs       Visual by character using v_CTRL-O in Select mode
V        Visual by line
Vs       Visual by line using v_CTRL-O in Select mode
CTRL-V   Visual blockwise
CTRL-Vs  Visual blockwise using v_CTRL-O in Select mode
s        Select by character
S        Select by line
CTRL-S   Select blockwise
i        Insert
ic       Insert mode completion compl-generic
ix       Insert mode i_CTRL-X completion
R        Replace R
Rc       Replace mode completion compl-generic
Rx       Replace mode i_CTRL-X completion
Rv       Virtual Replace gR
Rvc      Virtual Replace mode completion compl-generic
Rvx      Virtual Replace mode i_CTRL-X completion
c        Command-line editing
cr       Command-line editing overstrike mode c_<Insert>
cv       Non-interactive Ex mode -es
r        Hit-enter prompt
rm       The -- more -- prompt
r?       A :confirm query of some sort
!        Shell or external command is executing
t        Terminal mode: keys go to the job
```

This is useful in the 'statusline' option or RPC calls. In
most other places it always returns "c" or "n".
Note that in the future more modes and more specific modes may
be added. It's better not to compare the whole string but
only the leading character(s).
Also see visualmode().

Parameters:
- {expr} (any?)

Return:
(any)

msgpackdump({list} [, {type}]) msgpackdump()

Lua: Prefer vim.mpack.encode(); list wrapping, return shape,
and special-dict semantics differ.

Convert a list of Vimscript objects to msgpack. Returned value is a
readfile()-style list. When {type} contains "B", a Blob is
returned instead. Example:

```
call writefile(msgpackdump([{}]), 'fname.mpack', 'b')
```

or, using a Blob:

```
call writefile(msgpackdump([{}], 'B'), 'fname.mpack')
```

This will write the single 0x80 byte to a fname.mpack file
(dictionary with zero items is represented by 0x80 byte in
messagepack).

Limitations: E5004 E5005

1. Funcrefs cannot be dumped.

2. Containers that reference themselves cannot be dumped.

3. Dictionary keys are always dumped as STR strings.

4. Other strings and Blobs are always dumped as BIN strings.

5. Points 3. and 4. do not apply to msgpack-special-dicts.

Parameters:
- {list} (any)

- {type} (any?)

Return:
(any)

msgpackparse({data}) msgpackparse()

Lua: Prefer vim.mpack.decode(); input and special-dict
semantics differ.

Convert a readfile()-style list or a Blob to a list of
Vimscript objects.
Example:

```
let fname = expand('~/.config/nvim/shada/main.shada')
let mpack = readfile(fname, 'b')
let shada_objects = msgpackparse(mpack)
```

This will read ~/.config/nvim/shada/main.shada file to
shada_objects list.

Limitations:

1. Mapping ordering is not preserved unless messagepack
   mapping is dumped using generic mapping
   (msgpack-special-map).
2. Since the parser aims to preserve all data untouched
   (except for 1.) some strings are parsed to
   msgpack-special-dict format which is not convenient to
   use.

msgpack-special-dict

Some messagepack strings may be parsed to special
dictionaries. Special dictionaries are dictionaries which

3. Contain exactly two keys: _TYPE and _VAL.

4. _TYPE key is one of the types found in v:msgpack_types

variable.
5. Value for _VAL has the following format (Key column

contains name of the key from v:msgpack_types):

Key	Value
nil	Zero, ignored when dumping. Not returned by msgpackparse() since v:null was introduced.
boolean	One or zero. When dumping it is only checked that value is a Number. Not returned by msgpackparse() since v:true and v:false were introduced.

integer	List with four numbers: sign (-1 or 1), highest two bits, number with bits from 62nd to 31st, lowest 31 bits. I.e. to get actual number one will need to use code like
	`_VAL[0] * ((_VAL[1] << 62)`
	`& (_VAL[2] << 31)`
	`& _VAL[3])`

Special dictionary with this type will appear in
msgpackparse() output under one of the following
circumstances:
1. Number is 32-bit and value is either above

INT32_MAX or below INT32_MIN.
2. Number is 64-bit and value is above INT64_MAX. It

cannot possibly be below INT64_MIN because msgpack
C parser does not support such values.

float Float. This value cannot possibly appear in
msgpackparse() output.

string String, or Blob if binary string contains zero
byte. This value cannot appear in msgpackparse()
output since blobs were introduced.

array List. This value cannot appear in msgpackparse()
output.

msgpack-special-map

map List of Lists with two items (key and value) each.
This value will appear in msgpackparse() output if
parsed mapping contains one of the following keys:
3. Any key that is not a string (including keys which

are binary strings).
4. String with NUL byte inside.

5. Duplicate key.

ext List with two values: first is a signed integer
representing extension type. Second is
readfile()-style list of strings.

Parameters:
- {data} (any)

Return:
(any)

nextnonblank({lnum}) nextnonblank()

Return the line number of the first line at or below {lnum}
that is not blank. Example:

```
if getline(nextnonblank(1)) =~ "Java" | endif
```

When {lnum} is invalid or there is no non-blank line at or
below it, zero is returned.
{lnum} is used like with getline().
See also prevnonblank().

Parameters:
- {lnum} (integer|string)

Return:
(integer)

nr2char({expr} [, {utf8}]) nr2char()

Lua: Prefer string.char(): only works with ASCII.

Gets a UTF-8 string for a single codepoint {expr}.
Examples:

```
echo nr2char(64)                " returns '@'
echo nr2char(32)                " returns ' '
echo nr2char(300)               " returns I with bow character
```

UTF-8 encoding is always used, {utf8} option has no effect,
and exists only for backwards-compatibility.
Note that a NUL character in the file is specified with
nr2char(10), because NULs are represented with newline
characters. nr2char(0) is a real NUL and terminates the
string, thus results in an empty string.

Attributes:
api-fast

Parameters:
- {expr} (integer)

- {utf8} (boolean?)

Return:
(string)

nvim_...({...}) nvim_...() E5555 eval-api

Lua: Prefer vim.api.

Call nvim api functions. The type checking of arguments will
be stricter than for most other builtins. For instance,
if Integer is expected, a Number must be passed in, a
String will not be autoconverted.
Buffer numbers, as returned by bufnr() could be used as
first argument to nvim_buf_... functions. All functions
expecting an object (buffer, window or tabpage) can
also take the numerical value 0 to indicate the current
(focused) object.

Parameters:
- {...} (any)

Return:
(any)

or({expr}, {expr}) or()

Lua: Prefer bit.bor().

Bitwise OR on the two arguments. The arguments are converted
to a number. A List, Dict or Float argument causes an error.
Also see and() and xor().
Example:

```
let bits = or(bits, 0x80)
```

Rationale: The reason this is a function and not using the "|"
character like many languages, is that Vi has always used "|"
to separate commands. In many places it would not be clear if
"|" is an operator or a command separator.

Parameters:
- {expr} (number)

- {expr1} (number)

Return:
(any)

pathshorten({path} [, {len}]) pathshorten()

Shorten directory names in the path {path} and return the
result. The tail, the file name, is kept as-is. The other
components in the path are reduced to {len} letters in length.
If {len} is omitted or smaller than 1 then 1 is used (single
letters). Leading '~' and '.' characters are kept. Examples:

```
echo pathshorten('~/.config/nvim/autoload/file1.vim')
```

~/.c/n/a/file1.vim

```
echo pathshorten('~/.config/nvim/autoload/file2.vim', 2)
```

~/.co/nv/au/file2.vim
It doesn't matter if the path exists or not.
Returns an empty string on error.

Parameters:

- {path} (string)

- {len} (integer?)

Return:
(string)

perleval({expr}) perleval()

Evaluate perl expression {expr} and return its result
converted to Vim data structures.
Numbers and strings are returned as they are (strings are
copied though).
Lists are represented as Vim List type.
Dictionaries are represented as Vim Dictionary type,
non-string keys result in error.

Note: If you want an array or hash, {expr} must return a
reference to it.
Example:

```
echo perleval('[1 .. 4]')
```

[1, 2, 3, 4]

Parameters:
- {expr} (any)

Return:
(any)

pow({x}, {y}) pow()

Lua: Prefer math.pow().

Return the power of {x} to the exponent {y} as a Float.
{x} and {y} must evaluate to a Float or a Number.
Returns 0.0 if {x} or {y} is not a Float or a Number.
Examples:

```
echo pow(3, 3)
```

27.0

```
echo pow(2, 16)
```

65536.0

```
echo pow(32, 0.20)
```

2.0

Parameters:
- {x} (number)

- {y} (number)

Return:
(number)

preinserted() preinserted()

Returns non-zero if text has been inserted after the cursor
because "preinsert" is present in 'completeopt', or because
"longest" is present in 'completeopt' while 'autocomplete'
is active. Otherwise returns zero.

Return:
(number)

prevnonblank({lnum}) prevnonblank()

Return the line number of the first line at or above {lnum}
that is not blank. Example:

```
let ind = indent(prevnonblank(v:lnum - 1))
```

When {lnum} is invalid or there is no non-blank line at or
above it, zero is returned.
{lnum} is used like with getline().
Also see nextnonblank().

Parameters:
- {lnum} (integer|string)

Return:
(integer)

printf({fmt}, {expr1} ...) printf()

Lua: Prefer string.format().

Return a String with {fmt}, where "%" items are replaced by the formatted form of their respective arguments. Example:

```
echo printf("%4d: E%d %.30s", lnum, errno, msg)
```

May result in:

```
"  99: E42 asdfasdfasdfasdfasdfasdfasdfas"
```

When used as a method the base is passed as the second argument:

```
Compute()->printf("result: %d")
```

You can use call() to pass the items as a list.

Often used items are:

```
  %s    string
  %6S   string right-aligned in 6 display cells
  %6s   string right-aligned in 6 bytes
  %.9s  string truncated to 9 bytes
  %c    single byte
  %d    decimal number
  %5d   decimal number padded with spaces to 5 characters
  %b    binary number
  %08b  binary number padded with zeros to at least 8 characters
  %B    binary number using upper case letters
  %x    hex number
  %04x  hex number padded with zeros to at least 4 characters
  %X    hex number using upper case letters
  %o    octal number
  %f    floating point number as 12.23, inf, -inf or nan
  %F    floating point number as 12.23, INF, -INF or NAN
  %e    floating point number as 1.23e3, inf, -inf or nan
  %E    floating point number as 1.23E3, INF, -INF or NAN
  %g    floating point number, as %f or %e depending on value
  %G    floating point number, as %F or %E depending on value
  %%    the % character itself
  %p    representation of the pointer to the container
```

Conversion specifications start with '%' and end with the conversion type. All other characters are copied unchanged to the result.

The "%" starts a conversion specification. The following arguments appear in sequence:

```
        % [pos-argument] [flags] [field-width] [.precision] type
```

pos-argument
: At most one positional argument specifier. These take the form {n$}, where n is >= 1.

flags
: Zero or more of the following flags:

    #
    : The value should be converted to an "alternate form". For c, d, and s conversions, this option has no effect. For o conversions, the precision of the number is increased to force the first character of the output string to a zero (except if a zero value is printed with an explicit precision of zero).
    For x and X conversions, a non-zero result has the string "0x" (or "0X" for X conversions) prepended to it.

    0 (zero)
    : Zero padding. For all conversions the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (d, o, x, and X), the 0 flag is ignored.

    -
    : A negative field width flag; the converted value is to be left adjusted on the field boundary. The converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A - overrides a 0 if both are given.

    ' ' (space)
    : A blank should be left before a positive number produced by a signed conversion (d).

    +
    : A sign must always be placed before a number produced by a signed conversion. A + overrides a space if both are used.

field-width
: An optional decimal digit string specifying a minimum field width. If the converted value has fewer bytes than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given) to fill out the field width. For the S conversion the count is in cells.

.precision
        An optional precision, in the form of a period '.'
        followed by an optional digit string. If the digit
        string is omitted, the precision is taken as zero.
        This gives the minimum number of digits to appear for
        d, o, x, and X conversions, the maximum number of
        bytes to be printed from a string for s conversions,
        or the maximum number of cells to be printed from a
        string for S conversions.
        For floating point it is the number of digits after
        the decimal point.

type
        A character that specifies the type of conversion to
        be applied, see below.

A field width or precision, or both, may be indicated by an
asterisk "*" instead of a digit string. In this case, a
Number argument supplies the field width or precision. A
negative field width is treated as a left adjustment flag
followed by a positive field width; a negative precision is
treated as though it were missing. Example:

```
echo printf("%d: %.*s", nr, width, line)
```

This limits the length of the text used from "line" to
"width" bytes.

If the argument to be formatted is specified using a
positional argument specifier, and a '*' is used to indicate
that a number argument is to be used to specify the width or
precision, the argument(s) to be used must also be specified
using a {n$} positional argument specifier. See printf-$.

The conversion specifiers and their meanings are:

printf-d printf-b printf-B printf-o printf-x printf-X

dbBoxX The Number argument is converted to signed decimal (d),
        unsigned binary (b and B), unsigned octal (o), or
        unsigned hexadecimal (x and X) notation. The letters
        "abcdef" are used for x conversions; the letters
        "ABCDEF" are used for X conversions. The precision, if
        any, gives the minimum number of digits that must
        appear; if the converted value requires fewer digits, it
        is padded on the left with zeros. In no case does a
        non-existent or small field width cause truncation of a
        numeric field; if the result of a conversion is wider
        than the field width, the field is expanded to contain
        the conversion result.
        The 'h' modifier indicates the argument is 16 bits.
        The 'l' modifier indicates the argument is a long
        integer. The size will be 32 bits or 64 bits
        depending on your platform.
        The "ll" modifier indicates the argument is 64 bits.
        The b and B conversion specifiers never take a width
        modifier and always assume their argument is a 64 bit
        integer.
        Generally, these modifiers are not useful. They are
        ignored when type is known from the argument.

i   alias for d
D   alias for ld
U   alias for lu
O   alias for lo

printf-c

c   The Number argument is converted to a byte, and the
        resulting character is written.

printf-s

s   The text of the String argument is used. If a
        precision is specified, no more bytes than the number
        specified are used.
        If the argument is not a String type, it is
        automatically converted to text with the same format
        as ":echo".

printf-S

S   The text of the String argument is used. If a
        precision is specified, no more display cells than the
        number specified are used.

printf-f E807

f F   The Float argument is converted into a string of the
        form 123.456. The precision specifies the number of
        digits after the decimal point. When the precision is
        zero the decimal point is omitted. When the precision
        is not specified 6 is used. A really big number
        (out of range or dividing by zero) results in "inf"
         or "-inf" with %f (INF or -INF with %F).
         "0.0 / 0.0" results in "nan" with %f (NAN with %F).
        Example:

```
echo printf("%.2f", 12.115)
```

```
        12.12
```

Note that roundoff depends on the system libraries.
Use round() when in doubt.

printf-e printf-E

e E The Float argument is converted into a string of the form 1.234e+03 or 1.234E+03 when using 'E'. The precision specifies the number of digits after the decimal point, like with 'f'.

printf-g printf-G

g G The Float argument is converted like with 'f' if the value is between 0.001 (inclusive) and 10000000.0 (exclusive). Otherwise 'e' is used for 'g' and 'E' for 'G'. When no precision is specified superfluous zeroes and '+' signs are removed, except for the zero immediately after the decimal point. Thus 10000000.0 results in 1.0e7.

printf-%

% A '%' is written. No argument is converted. The complete conversion specification is "%%".

When a Number argument is expected a String argument is also accepted and automatically converted.
When a Float or String argument is expected a Number argument is also accepted and automatically converted.
Any other argument type results in an error message.

E766 E767

The number of {exprN} arguments must exactly match the number of "%" items. If there are not sufficient or too many arguments an error is given. Up to 18 arguments can be used.

printf-$

In certain languages, error and informative messages are more readable when the order of words is different from the corresponding message in English. To accommodate translations having a different word order, positional arguments may be used to indicate this. For instance:

```
#, c-format
msgid "%s returning %s"
msgstr "waarde %2$s komt terug van %1$s"
```

In this example, the sentence has its 2 string arguments reversed in the output.

```
echo printf(
    "In The Netherlands, vim's creator's name is: %1$s %2$s",
    "Bram", "Moolenaar")
```

```
    In The Netherlands, vim's creator's name is: Bram Moolenaar
```

```
echo printf(
    "In Belgium, vim's creator's name is: %2$s %1$s",
    "Bram", "Moolenaar")
```

```
    In Belgium, vim's creator's name is: Moolenaar Bram
```

Width (and precision) can be specified using the '*' specifier. In this case, you must specify the field width position in the argument list.

```
echo printf("%1$*2$.*3$d", 1, 2, 3)
```

```
    001
```

```
echo printf("%2$*3$.*1$d", 1, 2, 3)
```

```
      2
```

```
echo printf("%3$*1$.*2$d", 1, 2, 3)
```

```
    03
```

```
echo printf("%1$*2$.*3$g", 1.4142, 2, 3)
```

```
    1.414
```

You can mix specifying the width and/or precision directly and via positional arguments:

```
echo printf("%1$4.*2$f", 1.4142135, 6)
```

1.414214

```
echo printf("%1$*2$.4f", 1.4142135, 6)
```

1.4142

```
echo printf("%1$*2$.*3$f", 1.4142135, 6, 2)
```

1.41

You will get an overflow error E1510, when the field-width
or precision will result in a string longer than 1 MiB
(1024*1024 = 1048576) chars.

E1500

You cannot mix positional and non-positional arguments:

```
echo printf("%s%1$s", "One", "Two")
" E1500: Cannot mix positional and non-positional arguments:
" %s%1$s
```

E1501

You cannot skip a positional argument in a format string:

```
echo printf("%3$s%1$s", "One", "Two", "Three")
" E1501: format argument 2 unused in $-style format:
" %3$s%1$s
```

E1502

You can re-use a [field-width] (or [precision]) argument:

```
echo printf("%1$d at width %2$d is: %1$0*2$d", 1, 2)
" 1 at width 2 is: 01
```

However, you can't use it as a different type:

```
echo printf("%1$d at width %2$ld is: %1$0*2$d", 1, 2)
" E1502: Positional argument 2 used as field width reused as
" different type: long int/int
```

E1503

When a positional argument is used, but not the correct number
or arguments is given, an error is raised:

```
echo printf("%1$d at width %2$d is: %1$0*2$.*3$d", 1, 2)
" E1503: Positional argument 3 out of bounds: %1$d at width
" %2$d is: %1$0*2$.*3$d
```

Only the first error is reported:

```
echo printf("%1$0*2$.*3$d %4$d", 1, 2)
" E1503: Positional argument 3 out of bounds: %1$0*2$.*3$d
" %4$d
```

E1504

A positional argument can be used more than once:

```
echo printf("%1$s %2$s %1$s", "One", "Two")
" One Two One
```

However, you can't use a different type the second time:

```
echo printf("%1$s %2$s %1$d", "One", "Two")
" E1504: Positional argument 1 type used inconsistently:
" int/string
```

E1505

Various other errors that lead to a format string being
wrongly formatted lead to:

```
echo printf("%1$d at width %2$d is: %01$*2$.3$d", 1, 2)
" E1505: Invalid format specifier: %1$d at width %2$d is:
" %01$*2$.3$d
```

E1507

This internal error indicates that the logic to parse a
positional format argument ran into a problem that couldn't be
otherwise reported. Please file a bug against Vim if you run
into this, copying the exact format string and parameters that
were used.

Parameters:

- {fmt} (string)

- {expr1} (any?)

Return:
(string)

prompt_appendbuf({buf}, {text}) prompt_appendbuf()

Appends text to prompt buffer before current prompt. When {text} is a List: Append each item of the List as a text line above prompt-line in the buffer. Any type of item is accepted and converted to a String. Returns 1 for failure ({buf} not a prmopt buffer), 0 for success. When {text} is an empty list zero is returned.

Example:

```
func TextEntered(text)
  call prompt_appendbuf(bufnr(''), split('Entered: "' . a:text . '"', '\n'))
endfunc

set buftype=prompt
call prompt_setcallback(bufnr(''), function("TextEntered"))
eval bufnr("")->prompt_setprompt("cmd: ")
startinsert
```

Parameters:
- {buf} (integer|string)

- {text} (string|string[])

Return:
(0|1)

prompt_getinput({buf}) prompt_getinput()

Gets the current user-input in prompt-buffer {buf} without invoking prompt_callback. {buf} can be a buffer name or number.

If the buffer doesn't exist or isn't a prompt buffer, an empty string is returned.

Parameters:
- {buf} (integer|string)

Return:
(any)

prompt_getprompt({buf}) prompt_getprompt()

Returns the effective prompt text for buffer {buf}. {buf} can be a buffer name or number. See prompt-buffer.

If the buffer doesn't exist or isn't a prompt buffer, an empty string is returned.

Parameters:
- {buf} (integer|string)

Return:
(any)

prompt_setcallback({buf}, {expr}) prompt_setcallback()

Set prompt callback for buffer {buf} to {expr}. When {expr} is an empty string the callback is removed. This has only effect if {buf} has 'buftype' set to "prompt".

The callback is invoked when pressing Enter. The current buffer will always be the prompt buffer. A new line for a prompt is added before invoking the callback, thus the prompt for which the callback was invoked will be in the last but one line.
If the callback wants to add text to the buffer, it must insert it above the last line, since that is where the current prompt is. This can also be done asynchronously.
The callback is invoked with one argument, which is the text that was entered at the prompt. This can be an empty string if the user only typed Enter.
Example:

```
func s:TextEntered(text)
  if a:text == 'exit' || a:text == 'quit'
    stopinsert
    " Reset 'modified' to allow the buffer to be closed.
    " We assume there is nothing useful to be saved.
    set nomodified
    close
  else
    " Do something useful with "a:text".  In this example
    " we just repeat it.
    call append(line('$') - 1, 'Entered: "' .. a:text .. '"')
  endif
endfunc
call prompt_setcallback(bufnr(), function('s:TextEntered'))
```

Parameters:
- {buf} (integer|string)

- {expr} (string|function)

Return:
(any)

prompt_setinterrupt({buf}, {expr}) prompt_setinterrupt()

Set a callback for buffer {buf} to {expr}. When {expr} is an empty string the callback is removed. This has only effect if {buf} has 'buftype' set to "prompt".

This callback will be invoked when pressing CTRL-C in Insert mode. Without setting a callback Vim will exit Insert mode, as in any buffer.

Parameters:
- {buf} (integer|string)

- {expr} (string|function)

Return:
(any)

prompt_setprompt({buf}, {text}) prompt_setprompt()

Set prompt for buffer {buf} to {text}. You most likely want {text} to end in a space.
The result is only visible if {buf} has 'buftype' set to "prompt". Example:

```
call prompt_setprompt(bufnr(''), 'command: ')
```

Parameters:
- {buf} (integer|string)

- {text} (string)

Return:
(any)

pum_getpos() pum_getpos()

If the popup menu (see ins-completion-menu) is not visible, returns an empty Dictionary, otherwise, returns a Dictionary with the following keys:

height	nr of items visible
width	screen cells
row	top screen row (0 first row)
col	leftmost screen column (0 first col)
size	total nr of items
scrollbar	TRUE if scrollbar is visible

The values are the same as in v:event during CompleteChanged.

Return:
(any)

pumvisible() pumvisible()

Returns non-zero when the popup menu is visible, zero otherwise. See ins-completion-menu.
This can be used to avoid some things that would remove the popup menu.

Return:
(any)

py3eval({expr}) py3eval()

Evaluate Python expression {expr} and return its result

converted to Vim data structures.
Numbers and strings are returned as they are (strings are
copied though, Unicode strings are additionally converted to
UTF-8).
Lists are represented as Vim List type.
Dictionaries are represented as Vim Dictionary type with
keys converted to strings.

Parameters:
- {expr} (any)

Return:
(any)

pyeval({expr}) pyeval() E858 E859

Evaluate Python expression {expr} and return its result
converted to Vim data structures.
Numbers and strings are returned as they are (strings are
copied though).
Lists are represented as Vim List type.
Dictionaries are represented as Vim Dictionary type,
non-string keys result in error.

Parameters:
- {expr} (any)

Return:
(any)

pyxeval({expr}) pyxeval()

Evaluate Python expression {expr} and return its result
converted to Vim data structures.
Uses Python 2 or 3, see python x and 'pyxversion'.
See also: pyeval(), py3eval()

Parameters:
- {expr} (any)

Return:
(any)

rand([{expr}]) rand()

Lua: Prefer math.random().

Return a pseudo-random Number generated with an xoshiro128**
algorithm using seed {expr}. The returned number is 32 bits,
also on 64 bits systems, for consistency.
{expr} can be initialized by srand() and will be updated by
rand(). If {expr} is omitted, an internal seed value is used
and updated.
Returns -1 if {expr} is invalid.

Examples:

```
echo rand()
let seed = srand()
echo rand(seed)
echo rand(seed) % 16  " random number 0 - 15
```

Parameters:
- {expr} (number?)

Return:
(any)

range({expr} [, {max} [, {stride}]]) range() E726 E727

Returns a List with Numbers:
- If only {expr} is specified: [0, 1, ..., {expr} - 1]
- If {max} is specified: [{expr}, {expr} + 1, ..., {max}]
- If {stride} is specified: [{expr}, {expr} + {stride}, ...,

  {max}] (increasing {expr} with {stride} each time, not
  producing a value past {max}).
When the maximum is one before the start the result is an
empty list. When the maximum is more than one before the
start this is an error.
Examples:

```
echo range(4)           " [0, 1, 2, 3]
echo range(2, 4)        " [2, 3, 4]
echo range(2, 9, 3)     " [2, 5, 8]
echo range(2, -2, -1)   " [2, 1, 0, -1, -2]
echo range(0)           " []
echo range(2, 0)        " error!
```

Parameters:

- {expr} (any)
- {max} (integer?)
- {stride} (integer?)

Return:

(any)

readblob({fname} [, {offset} [, {size}]]) readblob()

Lua: Prefer io.open() with :read("*a") in binary mode for whole-file reads; offset, size, and Blob return differ.

Read file {fname} in binary mode and return a Blob.
If {offset} is specified, read the file from the specified offset. If it is a negative value, it is used as an offset from the end of the file. E.g., to read the last 12 bytes:

```
echo readblob('file.bin', -12)
```

If {size} is specified, only the specified size will be read. E.g. to read the first 100 bytes of a file:

```
echo readblob('file.bin', 0, 100)
```

If {size} is -1 or omitted, the whole data starting from {offset} will be read.
This can be also used to read the data from a character device on Unix when {size} is explicitly set. Only if the device supports seeking {offset} can be used. Otherwise it should be zero. E.g. to read 10 bytes from a serial console:

```
echo readblob('/dev/ttyS0', 0, 10)
```

When the file can't be opened an error message is given and the result is an empty Blob.
When the offset is beyond the end of the file the result is an empty blob.
When trying to read more bytes than are available the result is truncated.
Also see readfile() and writefile().

Parameters:

- {fname} (string)
- {offset} (integer?)
- {size} (integer?)

Return:

(any)

readdir({directory} [, {expr}]) readdir()

Lua: Prefer vim.fs.dir() as an iterator, then collect and filter results as needed.

Return a list with file and directory names in {directory}.
You can also use glob() if you don't need to do complicated things, such as limiting the number of matches.

When {expr} is omitted all entries are included.
When {expr} is given, it is evaluated to check what to do:
- If {expr} results in -1 then no further entries will be handled.
- If {expr} results in 0 then this entry will not be added to the list.
- If {expr} results in 1 then this entry will be added to the list.

Each time {expr} is evaluated v:val is set to the entry name.
When {expr} is a function the name is passed as the argument.
For example, to get a list of files ending in ".txt":

```
echo readdir(dirname, {n -> n =~ '\.txt$'})
```

To skip hidden and backup files:

```
echo readdir(dirname, {n -> n !~ '^\.\|\~$'})
```

If you want to get a directory tree:

```
function! s:tree(dir)
    return {a:dir : map(readdir(a:dir),
    \ {_, x -> isdirectory(x) ?
    \          {x : s:tree(a:dir .. '/' .. x)} : x})}
endfunction
echo s:tree(".")
```

Returns an empty List on error.

Parameters:

- {directory} (string)
- {expr} (integer|string|fun(name: string): integer?)

Return:

(any)

readfile({fname} [, {type} [, {max}]]) readfile()

Lua: Prefer io.open() with :read("*a"), then split lines as needed; {type}, {max}, and Blob return differ.

Read file {fname} and return a List, each line of the file as an item. Lines are broken at NL characters. Macintosh files separated with CR will result in a single long line (unless a NL appears somewhere).
All NUL characters are replaced with a NL character.
When {type} contains "b" binary mode is used:

- When the last line ends in a NL an extra empty list item is added.
- No CR characters are removed.

Otherwise:

- CR characters that appear before a NL are removed.
- Whether the last line ends in a NL or not does not matter.
- Any UTF-8 byte order mark is removed from the text.

When {max} is given this specifies the maximum number of lines to be read. Useful if you only want to check the first ten lines of a file:

```
for line in readfile(fname, '', 10)
  if line =~ 'Date' | echo line | endif
endfor
```

When {max} is negative -{max} lines from the end of the file are returned, or as many as there are.
When {max} is zero the result is an empty list.
Note that without {max} the whole file is read into memory.
Also note that there is no recognition of encoding. Read a file into a buffer if you need to.
Deprecated (use readblob() instead): When {type} contains "B" a Blob is returned with the binary data of the file unmodified.
When the file can't be opened an error message is given and the result is an empty list.
Also see writefile().

Parameters:

- {fname} (string)
- {type} (string?)
- {max} (integer?)

Return:

(string[])

reduce({object}, {func} [, {initial}]) reduce() E998

Lua: Prefer Iter:fold().

{func} is called for every item in {object}, which can be a String, List or a Blob. {func} is called with two

arguments: the result so far and current item. After
processing all items the result is returned. E1132

{initial} is the initial result. When omitted, the first item
in {object} is used and {func} is first called for the second
item. If {initial} is not given and {object} is empty no
result can be computed, an E998 error is given.

Examples:

```
echo reduce([1, 3, 5], { acc, val -> acc + val })
echo reduce(['x', 'y'], { acc, val -> acc .. val }, 'a')
echo reduce(0z1122, { acc, val -> 2 * acc + val })
echo reduce('xyz', { acc, val -> acc .. ',' .. val })
```

Parameters:
- {object} (any)
- {func} (fun(accumulator: T, current: any): any)
- {initial} (any?)

Return:
(T)

reg_executing() reg_executing()

Returns the single letter name of the register being executed.
Returns an empty string when no register is being executed.
See @.

Return:
(any)

reg_recorded() reg_recorded()

Returns the single letter name of the last recorded register.
Returns an empty string when nothing was recorded yet.
See q and Q.

Return:
(any)

reg_recording() reg_recording()

Returns the single letter name of the register being recorded.
Returns an empty string when not recording. See q.

Return:
(any)

reltime() reltime()

reltime({start})
reltime({start}, {end})
Lua: Prefer uv.hrtime().

Return an item that represents a time value. The item is a
list with items that depend on the system.
The item can be passed to reltimestr() to convert it to a
string or reltimefloat() to convert to a Float.

Without an argument it returns the current "relative time", an
implementation-defined value meaningful only when used as an
argument to reltime(), reltimestr() and reltimefloat().

With one argument it returns the time passed since the time
specified in the argument.
With two arguments it returns the time passed between {start}
and {end}.

The {start} and {end} arguments must be values returned by
reltime(). Returns zero on error.

Note: localtime() returns the current (non-relative) time.

Attributes:
api-fast

Parameters:
- {start} (any?)
- {end} (any?)

Return:
(any)

reltimefloat({time}) reltimefloat()

Lua: Prefer uv.hrtime() with (finish - start) / 1e9 for
elapsed seconds.

Return a Float that represents the time value of {time}.
Unit of time is seconds.

Example:
```
	let start = reltime()
	call MyFunction()
	let seconds = reltimefloat(reltime(start))
```
See the note of reltimestr() about overhead.
Also see profiling.
If there is an error an empty string is returned

Attributes:
api-fast

Parameters:
- {time} (any)

Return:
(any)

reltimestr({time}) reltimestr()

Return a String that represents the time value of {time}.
This is the number of seconds, a dot and the number of
microseconds. Example:

```
let start = reltime()
call MyFunction()
echo reltimestr(reltime(start))
```

Note that overhead for the commands will be added to the time.
Leading spaces are used to make the string align nicely. You
can use split() to remove it.

```
echo split(reltimestr(reltime(start)))[0]
```

Also see profiling.
If there is an error an empty string is returned

Attributes:
api-fast

Parameters:
- {time} (any)

Return:
(any)

remove({list}, {idx}) remove()

Lua: Prefer table.remove().

remove({list}, {idx}, {end})
Without {end}: Remove the item at {idx} from List {list} and
return the item.
With {end}: Remove items from {idx} to {end} (inclusive) and
return a List with these items. When {idx} points to the same
item as {end} a list with one item is returned. When {end}
points to an item before {idx} this is an error.
See list-index for possible values of {idx} and {end}.
Returns zero on error.
Example:

```
echo "last item: " .. remove(mylist, -1)
call remove(mylist, 0, 9)
```

Use delete() to remove a file.

Parameters:
- {list} (any[])
- {idx} (integer)
- {end} (integer?)

Return:
(any)

remove({blob}, {idx})
remove({blob}, {idx}, {end})
Without {end}: Remove the byte at {idx} from Blob {blob} and
return the byte.
With {end}: Remove bytes from {idx} to {end} (inclusive) and
return a Blob with these bytes. When {idx} points to the same
byte as {end} a Blob with one byte is returned. When {end}
points to a byte before {idx} this is an error.
Returns zero on error.
Example:

```
echo "last byte: " .. remove(myblob, -1)
call remove(mylist, 0, 9)
```

Parameters:
- {blob} (any)

- {idx} (integer)

- {end} (integer?)

Return:
(any)

remove({dict}, {key})
Lua: Prefer local value = dict[key]; dict[key] = nil for dicts; missing-key behavior differs.

Remove the entry from {dict} with key {key} and return it. Example:

```
echo "removed " .. remove(dict, "one")
```

If there is no {key} in {dict} this is an error. Returns zero on error.

Parameters:
- {dict} (any)

- {key} (string)

Return:
(any)

rename({from}, {to}) rename()

Lua: Prefer uv.fs_rename() or os.rename().

Rename the file by the name {from} to the name {to}. This should also work to move files across file systems. The result is a Number, which is 0 if the file was renamed successfully, and non-zero when the renaming failed. **NOTE:** If {to} exists it is overwritten without warning. This function is not available in the sandbox.

Parameters:
- {from} (string)

- {to} (string)

Return:
(integer)

repeat({expr}, {count}) repeat()

Lua: Prefer string.rep() for strings; lists need manual repetition.

Repeat {expr} {count} times and return the concatenated result. Example:

```
let separator = repeat('-', 80)
```

When {count} is zero or negative the result is empty. When {expr} is a List or a Blob the result is {expr} concatenated {count} times. Example:

```
let longlist = repeat(['a', 'b'], 3)
```

Results in ['a', 'b', 'a', 'b', 'a', 'b'].

Attributes:
api-fast

Parameters:
- {expr} (any)

- {count} (integer)

Return:
(any)

resolve({filename}) resolve() E655
On MS-Windows, when {filename} is a shortcut (a .lnk file), returns the path the shortcut points to in a simplified form. On Unix, repeat resolving symbolic links in all path components of {filename} and return the simplified result. To cope with link cycles, resolving of symbolic links is stopped after 100 iterations.

On other systems, return the simplified {filename}.
The simplification step is done as by simplify().
resolve() keeps a leading path component specifying the
current directory (provided the result is still a relative
path name) and also keeps a trailing path separator.

Attributes:
api-fast

Parameters:
- {filename} (string)

Return:
(string)

reverse({object}) reverse()

Lua: Prefer vim.iter(list):rev():totable() for lists;
strings, blobs, and in-place behavior differ.

Reverse the order of items in {object}. {object} can be a
List, a Blob or a String. For a List and a Blob the
items are reversed in-place and {object} is returned.
For a String a new String is returned.
Returns zero if {object} is not a List, Blob or a String.
If you want a List or Blob to remain unmodified make a copy
first:

```
let revlist = reverse(copy(mylist))
```

Parameters:
- {object} (T[])

Return:
(T[])

round({expr}) round()

Lua: Prefer
expr >= 0 and math.floor(expr + 0.5) or math.ceil(expr - 0.5)
for half-away-from-zero rounding.

Round off {expr} to the nearest integral value and return it
as a Float. If {expr} lies halfway between two integral
values, then use the larger one (away from zero).
{expr} must evaluate to a Float or a Number.
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo round(0.456)
```

```
0.0
```

```
echo round(4.5)
```

```
5.0
```

```
echo round(-4.5)
```

```
-5.0
```

Parameters:
- {expr} (number)

Return:
(number)

rpcnotify({channel}, {event} [, {args}...]) rpcnotify()

Lua: Prefer vim.rpcnotify().

Sends {event} to {channel} via RPC and returns immediately.
If {channel} is 0, the event is broadcast to all channels.
Example:

```
au VimLeave call rpcnotify(0, "leaving")
```

Parameters:
- {channel} (integer)
- {event} (string)
- {...} (any)

Return:
(integer)

rpcrequest({channel}, {method} [, {args}...]) rpcrequest()

Lua: Prefer vim.rpcrequest().

Sends a request to {channel} to invoke {method} via
RPC and blocks until a response is received.
Example:

```
let result = rpcrequest(rpc_chan, "func", 1, 2, 3)
```

Parameters:
- {channel} (integer)
- {method} (string)
- {...} (any)

Return:
(any)

rubyeval({expr}) rubyeval()

Evaluate Ruby expression {expr} and return its result
converted to Vim data structures.
Numbers, floats and strings are returned as they are (strings
are copied though).
Arrays are represented as Vim List type.
Hashes are represented as Vim Dictionary type.
Other objects are represented as strings resulted from their
"Object#to_s" method.

Parameters:
- {expr} (any)

Return:
(any)

screenattr({row}, {col}) screenattr()

Like screenchar(), but return the attribute. This is a rather
arbitrary number that can only be used to compare to the
attribute at other positions.
Returns -1 when row or col is out of range.

Parameters:
- {row} (integer)
- {col} (integer)

Return:
(integer)

screenchar({row}, {col}) screenchar()

Returns the character at screen position [row, col] as a
Number. This works for every possible screen position, also
status lines, window separators and the command line. The
top left position is row one, column one. The character
excludes composing characters. For double-byte
encodings it may only be the first byte.
This is mainly to be used for testing.
Returns -1 when row or col is out of range.

Parameters:
- {row} (integer)
- {col} (integer)

Return:
(integer)

screenchars({row}, {col}) screenchars()

Returns the character and any composing characters at screen
position [row, col] as a List of Numbers. The first number
is the same as what screenchar() returns; further numbers
are composing characters on top of the base character.
This is mainly to be used for testing.
Returns an empty List when row or col is out of range.

Parameters:
- {row} (integer)

- {col} (integer)

Return:
(integer[])

screencol() screencol()

Returns the current screen column of the cursor. The
leftmost column has number 1.
This function is mainly used for testing.

Note: Always returns the current screen column, thus if used
in a command (e.g. ":echo screencol()") it will return the
column inside the command line, which is 1 when the command is
executed. To get the cursor position in the file use one of
the following mappings:

```
nnoremap <expr> GG ":echom " .. screencol() .. "\n"
nnoremap <silent> GG :echom screencol()<CR>
noremap GG <Cmd>echom screencol()<CR>
```

Return:
(integer)

screenpos({winid}, {lnum}, {col}) screenpos()

Returns the screen position of the text character in window
{winid} at buffer line {lnum} and column {col} as a Dict.
{col} is a one-based byte index.
The Dict has these members:
row screen row
col first screen column
endcol last screen column
curscol cursor screen column
If the specified position is not visible, all values are zero.
The "endcol" value differs from "col" when the character
occupies more than one screen cell. E.g. for a Tab "col" can
be 1 and "endcol" can be 8.
The "curscol" value is where the cursor would be placed. For
a Tab it would be the same as "endcol", while for a double
width character it would be the same as "col".
The conceal feature is ignored here, the column numbers are
as if 'conceallevel' is zero. You can set the cursor to the
right position and use screencol() to get the value with
conceal taken into account.
If the position is in a closed fold the screen position of the
first character is returned, {col} is not used.
Returns an empty Dict if {winid} is invalid.

Parameters:
- {winid} (integer)

- {lnum} (integer)

- {col} (integer)

Return:
({ col: integer, curscol: integer, endcol: integer, row: integer })

screenrow() screenrow()

Returns the current screen row of the cursor. The top line
has number one.
This function is mainly used for testing.
Alternatively you can use winline().

Note: Same restrictions as with screencol().

Return:
(integer)

screenstring({row}, {col}) screenstring()

Returns the base character and any composing characters at
screen position [row, col] as a String. This is like
screenchars() but returning a String with the characters.
This is mainly to be used for testing.
Returns an empty String when row or col is out of range.

Parameters:
- {row} (integer)

- {col} (integer)

Return:
(string)

search({pattern} [, {flags} [, {stopline} [, {timeout} [, {skip}]]]]) search()
Search for regexp pattern {pattern}. The search starts at the cursor position (you can use cursor() to set it).

When a match has been found its line number is returned. If there is no match a 0 is returned and the cursor doesn't move. No error message is given. To get the matched string, use matchbufline().

{flags} is a String, which can contain these character flags:
'b' search Backward instead of forward
'c' accept a match at the Cursor position
'e' move to the End of the match
'n' do Not move the cursor
'p' return number of matching sub-Pattern (see below)
's' Set the ' mark at the previous location of the cursor
'w' Wrap around the end of the file
'W' don't Wrap around the end of the file
'z' start searching at the cursor column instead of Zero
If neither 'w' or 'W' is given, the 'wrapscan' option applies.

If the 's' flag is supplied, the ' mark is set, only if the cursor is moved. The 's' flag cannot be combined with the 'n' flag.

'ignorecase', 'smartcase' and 'magic' are used.

When the 'z' flag is not given, forward searching always starts in column zero and then matches before the cursor are skipped. When the 'c' flag is present in 'cpo' the next search starts after the match. Without the 'c' flag the next search starts one column after the start of the match. This matters for overlapping matches. See cpo-c. You can also insert "\ze" to change where the match ends, see /\ze.

When searching backwards and the 'z' flag is given then the search starts in column zero, thus no match in the current line will be found (unless wrapping around the end of the file).

When the {stopline} argument is given then the search stops after searching this line. This is useful to restrict the search to a range of lines. Examples:

```
let match = search('(', 'b', line("w0"))
let end = search('END', '', line("w$"))
```

When {stopline} is used and it is not zero this also implies that the search does not wrap around the end of the file. A zero value is equal to not giving the argument.

When the {timeout} argument is given the search stops when more than this many milliseconds have passed. Thus when {timeout} is 500 the search stops after half a second. The value must not be negative. A zero value is like not giving the argument.

Note: the timeout is only considered when searching, not while evaluating the {skip} expression.

If the {skip} expression is given it is evaluated with the cursor positioned on the start of a match. If it evaluates to non-zero this match is skipped. This can be used, for example, to skip a match in a comment or a string. {skip} can be a string, which is evaluated as an expression, a function reference or a lambda. When {skip} is omitted or empty, every match is accepted. When evaluating {skip} causes an error the search is aborted and -1 returned.

search()-sub-match

With the 'p' flag the returned value is one more than the first sub-match in \(\). One if none of them matched but the whole pattern did match. To get the column number too use searchpos().

The cursor will be positioned at the match, unless the 'n' flag is used.

Example (goes over all files in the argument list):

```
let n = 1
while n <= argc()           " loop over all files in arglist
  exe "argument " .. n
  " start at the last char in the file and wrap for the
  " first search to find match at start of file
  normal G$
  let flags = "w"
  while search("foo", flags) > 0
    s/foo/bar/g
    let flags = "W"
  endwhile
  update                    " write the file if modified
  let n = n + 1
endwhile
```

Example for using some flags:

```
echo search('\<if\|\(else\)\|\(endif\)', 'ncpe')
```

This will search for the keywords "if", "else", and "endif"
under or after the cursor. Because of the 'p' flag, it
returns 1, 2, or 3 depending on which keyword is found, or 0
if the search fails. With the cursor on the first word of the
line:

```
if (foo == 0) | let foo = foo + 1 | endif
```

the function returns 1. Without the 'c' flag, the function
finds the "endif" and returns 3. The same thing happens
without the 'e' flag if the cursor is on the "f" of "if".
The 'n' flag tells the function not to move the cursor.

Parameters:

- {pattern} (string)
- {flags} (string?)
- {stopline} (integer?)
- {timeout} (integer?)
- {skip} (string|function?)

Return:

(integer)

searchcount([{options}]) searchcount()

Get or update the last search count, like what is displayed
without the "S" flag in 'shortmess'. This works even if
'shortmess' does contain the "S" flag.

This returns a Dictionary. The dictionary is empty if the
previous pattern was not set and "pattern" was not specified.

key	type	meaning
current	Number	current position of match; 0 if the cursor position is before the first match
exact_match	Boolean	1 if "current" is matched on "pos", otherwise 0
total	Number	total count of matches found
incomplete	Number	0: search was fully completed 1: recomputing was timed out 2: max count exceeded

For {options} see further down.

To get the last search count when n or N was pressed, call
this function with recompute: 0 . This sometimes returns
wrong information because of 'maxsearchcount'.
If the count exceeded 'maxsearchcount', the result must be
'maxsearchcount' + 1. If you want to get correct information,
specify recompute: 1:

```
" result == 'maxsearchcount' + 1 when many matches
let result = searchcount(#{recompute: 0})

" Below returns correct result (recompute defaults
" to 1)
let result = searchcount()
```

The function is useful to add the count to 'statusline':

```
function! LastSearchCount() abort
  let result = searchcount(#{recompute: 0})
  if empty(result)
    return ''
  endif
  if result.incomplete ==# 1     " timed out
    return printf(' /%s [?/??]', @/)
  elseif result.incomplete ==# 2 " max count exceeded
    if result.total > result.maxcount &&
    \  result.current > result.maxcount
      return printf(' /%s [>%d/>%d]', @/,
      \             result.current, result.total)
    elseif result.total > result.maxcount
      return printf(' /%s [%d/>%d]', @/,
      \             result.current, result.total)
    endif
  endif
  return printf(' /%s [%d/%d]', @/,
  \             result.current, result.total)
endfunction
let &statusline ..= '%{LastSearchCount()}'

" Or if you want to show the count only when
" 'hlsearch' was on
" let &statusline ..=
" \   '%{v:hlsearch ? LastSearchCount() : ""}'
```

You can also update the search count, which can be useful in a CursorMoved or CursorMovedI autocommand:

```
autocmd CursorMoved,CursorMovedI *
  \ let s:searchcount_timer = timer_start(
  \   200, function('s:update_searchcount'))
function! s:update_searchcount(timer) abort
  if a:timer ==# s:searchcount_timer
    call searchcount(#{
    \ recompute: 1, maxcount: 0, timeout: 100})
    redrawstatus
  endif
endfunction
```

This can also be used to count matched texts with specified pattern in the current buffer using "pattern":

```
" Count '\<foo\>' in this buffer
" (Note that it also updates search count)
let result = searchcount(#{pattern: '\<foo\>'})

" To restore old search count by old pattern,
" search again
call searchcount()
```

{options} must be a Dictionary. It can contain:

key	type	meaning
recompute	Boolean	if TRUE, recompute the count like n or N was executed. otherwise returns the last computed result (when n or N was used when "S" is not in 'shortmess', or this function was called). (default: TRUE)
pattern	String	recompute if this was given and different with @/. this works as same as the below command is executed before calling this function

```
let @/ = pattern
```

		(default: @/)
timeout	Number	0 or negative number is no timeout. timeout milliseconds for recomputing the result (default: 0)
maxcount	Number	0 or negative number is no limit. max count of matched text while recomputing the result. if search exceeded total count, "total" value becomes maxcount + 1 (default: 'maxsearchcount')
pos	List	[lnum, col, off] value when recomputing the result. this changes "current" result value. see cursor(), getpos() (default: cursor's position)

Parameters:
- {options} (table?)

Return:
(any)

searchdecl({name} [, {global} [, {thisblock}]]) searchdecl()

Search for the declaration of {name}.

With a non-zero {global} argument it works like gD, find first match in the file. Otherwise it works like gd, find first match in the function.

With a non-zero {thisblock} argument matches in a {} block that ends before the cursor position are ignored. Avoids finding variable declarations only valid in another scope.

Moves the cursor to the found match.
Returns zero for success, non-zero for failure.
Example:

```
if searchdecl('myvar') == 0
   echo getline('.')
endif
```

Parameters:
- {name} (string)
- {global} (boolean?)
- {thisblock} (boolean?)

Return:
(any)

searchpair()

searchpair({start}, {middle}, {end} [, {flags} [, {skip} [, {stopline} [, {timeout}]]]])
Search for the match of a nested start-end pair. This can be used to find the "endif" that matches an "if", while other if/endif pairs in between are ignored.
The search starts at the cursor. The default is to search forward, include 'b' in {flags} to search backward.
If a match is found, the cursor is positioned at it and the line number is returned. If no match is found 0 or -1 is returned and the cursor doesn't move. No error message is given.

{start}, {middle} and {end} are patterns, see pattern. They must not contain \(\) pairs. Use of \%(\) is allowed. When {middle} is not empty, it is found when searching from either direction, but only when not in a nested start-end pair. A typical use is:

```
echo searchpair('\<if\>', '\<else\>', '\<endif\>')
```

By leaving {middle} empty the "else" is skipped.

{flags} 'b', 'c', 'n', 's', 'w' and 'W' are used like with search(). Additionally:
'r' Repeat until no more matches found; will find the outer pair. Implies the 'W' flag.
'm' Return number of matches instead of line number with the match; will be > 1 when 'r' is used.
Note: it's nearly always a good idea to use the 'W' flag, to avoid wrapping around the end of the file.

When a match for {start}, {middle} or {end} is found, the {skip} expression is evaluated with the cursor positioned on the start of the match. It should return non-zero if this match is to be skipped. E.g., because it is inside a comment or a string.
When {skip} is omitted or empty, every match is accepted.
When evaluating {skip} causes an error the search is aborted and -1 returned.
{skip} can be a string, a lambda, a funcref or a partial.
Anything else makes the function fail.

For {stopline} and {timeout} see search().

The value of 'ignorecase' is used. 'magic' is ignored, the patterns are used like it's on.

The search starts exactly at the cursor. A match with {start}, {middle} or {end} at the next character, in the direction of searching, is the first one found. Example:

```
if 1
  if 2
  endif 2
endif 1
```

When starting at the "if 2", with the cursor on the "i", and searching forwards, the "endif 2" is found. When starting on the character just before the "if 2", the "endif 1" will be found. That's because the "if 2" will be found first, and then this is considered to be a nested if/endif from "if 2" to "endif 2".
When searching backwards and {end} is more than one character, it may be useful to put "\zs" at the end of the pattern, so that when the cursor is inside a match with the end it finds the matching start.

Example, to find the "endif" command in a Vim script:

```
echo searchpair('\<if\>', '\<el\%[seif]\>', '\<en\%[dif]\>', 'W',
\ 'getline(".") =~ "^\\s*\""')
```

The cursor must be at or after the "if" for which a match is to be found. Note that single-quote strings are used to avoid having to double the backslashes. The skip expression only catches comments at the start of a line, not after a command. Also, a word "en" or "if" halfway through a line is considered a match.
Another example, to search for the matching "{" of a "}":

```
echo searchpair('{', '', '}', 'bW')
```

This works when the cursor is at or before the "}" for which a match is to be found. To reject matches that syntax highlighting recognized as strings:

```
echo searchpair('{', '', '}', 'bW',
     \ 'synIDattr(synID(line("."), col("."), 0), "name") =~? "string"')
```

Parameters:

- {start} (string)
- {middle} (string)
- {end} (string)
- {flags} (string?)
- {skip} (string|function?)
- {stopline} (integer?)
- {timeout} (integer?)

Return:
(integer)

searchpairpos()

searchpairpos({start}, {middle}, {end} [, {flags} [, {skip} [, {stopline} [, {timeout}]]]])
Same as searchpair(), but returns a List with the line and column position of the match. The first element of the List is the line number and the second element is the byte index of the column position of the match. If no match is found, returns [0, 0].

```
let [lnum,col] = searchpairpos('{', '', '}', 'n')
```

See match-parens for a bigger and more useful example.

Parameters:

- {start} (string)
- {middle} (string)
- {end} (string)

- {flags} (string?)
- {skip} (string|function?)
- {stopline} (integer?)
- {timeout} (integer?)

Return:
([integer, integer])

searchpos()

searchpos({pattern} [, {flags} [, {stopline} [, {timeout} [, {skip}]]]])
Same as search(), but returns a List with the line and column position of the match. The first element of the List is the line number and the second element is the byte index of the column position of the match. If no match is found, returns [0, 0].
Example:

```
let [lnum, col] = searchpos('mypattern', 'n')
```

When the 'p' flag is given then there is an extra item with the sub-pattern match number search()-sub-match. Example:

```
let [lnum, col, submatch] = searchpos('\(\l\)\|\(\u\)', 'np')
```

In this example "submatch" is 2 when a lowercase letter is found /\l, 3 when an uppercase letter is found /\u.

Parameters:
- {pattern} (string)
- {flags} (string?)
- {stopline} (integer?)
- {timeout} (integer?)
- {skip} (string|function?)

Return:
({ [1]: integer, [2]: integer, [3]: integer? })

serverlist([{opts}])

serverlist()

Returns a list of server addresses, or empty if all servers were stopped. serverstart() serverstop()

The optional argument {opts} is a Dict and supports the following items:

peer : If TRUE, servers not started by serverstart() will also be returned. (default: FALSE)
Not supported on Windows yet.

info : If TRUE, return a list of Dicts with detailed info instead of addresses. Implies peer=true. Each Dict has the following items:

addr	(string)	Server address.
pid	(number)	PID of the Nvim process.
own	(bool)	Whether this server belongs to the current Nvim instance.
active	(number)	v:useractive of the server.

(default: FALSE)

Example:

```
echo serverlist()
echo serverlist(#{info: v:true})
```

Parameters:
- {opts} (table?)

Return:
(string[]|vim.ServerInfo[])

serverstart([{address}])

serverstart()

Opens a socket or named pipe at {address} and listens for RPC messages. Clients can send API commands to the returned address to control Nvim.

Returns the address string (which may differ from the
{address} argument, see below).

- If {address} has a colon (":") it is a TCP/IPv4/IPv6 address

  where the last ":" separates host and port (empty or zero
  assigns a random port).
- Else {address} is the path to a named pipe (except on Windows).

  - If {address} has no slashes ("/") it is treated as the

    "name" part of a generated path in this format:

    ```
    stdpath("run")."/{name}.{pid}.{counter}"
    ```

  - If {address} is omitted the name is "nvim".

    ```
    echo serverstart()
    ```

    ```
    => /tmp/nvim.bram/oknANW/nvim.15430.5
    ```

Example bash command to list all Nvim servers:

```
ls ${XDG_RUNTIME_DIR:-${TMPDIR}nvim.${USER}}/*/nvim.*.0
```

Example named pipe:

```
if has('win32')
  echo serverstart('//./pipe/nvim-pipe-1234')
else
  echo serverstart('nvim.sock')
endif
```

Example TCP/IP address:

```
echo serverstart('::1:12345')
```

Parameters:
- {address} (string?)

Return:
(string)

serverstop({address}) serverstop()

Closes the pipe or socket at {address}.
Returns TRUE if {address} is valid, else FALSE.
If v:servername is stopped it is set to the next available
address in serverlist().

Parameters:
- {address} (string)

Return:
(integer)

setbufline({buf}, {lnum}, {text}) setbufline()

Lua: Prefer nvim_buf_set_lines().

Set line {lnum} to {text} in buffer {buf}. This works like
setline() for the specified buffer.

This function works only for loaded buffers. First call
bufload() if needed.

To insert lines use appendbufline().

{text} can be a string to set one line, or a List of strings
to set multiple lines. If the List extends below the last
line then those lines are added. If the List is empty then
nothing is changed and zero is returned.

For the use of {buf}, see bufname() above.

{lnum} is used like with setline().
Use "$" to refer to the last line in buffer {buf}.
When {lnum} is just below the last line the {text} will be
added below the last line.
On success 0 is returned, on failure 1 is returned.

If {buf} is not a valid buffer or {lnum} is not valid, an
error message is given.

Parameters:
- {buf} (integer|string)

- {lnum} (integer)

- {text} (string|string[])

Return:
(integer)

setbufvar({buf}, {varname}, {val}) setbufvar()

Lua: Prefer nvim_buf_set_var() or vim.b after resolving {buf} to a bufnr; option names use nvim_set_option_value().

Set option or local variable {varname} (string, without "b:") in buffer {buf} to {val}. Also works for a global or window-local option (not variable). A window-local option is set in a window of the current tabpage that displays {buf}, and the global value is unchanged. When no window in the current tabpage displays {buf} the option is not set and no error is given; use setwinvar() or win_execute() for that.

For the use of {buf}, see bufname() above.

Examples:

```
call setbufvar(1, "&mod", 1)
call setbufvar("todo", "myvar", "foobar")
```

This function is not available in the sandbox.

Parameters:
- {buf} (integer|string)

- {varname} (string)

- {val} (any)

Return:
(any)

setcellwidths({list}) setcellwidths()

Specify overrides for cell widths of character ranges. This tells Vim how wide characters are when displayed in the terminal, counted in screen cells. The values override 'ambiwidth'. Example:

```
call setcellwidths([
            \ [0x111, 0x111, 1],
            \ [0x2194, 0x2199, 2],
            \ ])
```

The {list} argument is a List of Lists with each three numbers: [{low}, {high}, {width}]. E1109 E1110
{low} and {high} can be the same, in which case this refers to one character. Otherwise it is the range of characters from {low} to {high} (inclusive). E1111 E1114
Only characters with value 0x80 and higher can be used.

{width} must be either 1 or 2, indicating the character width in screen cells. E1112

An error is given if the argument is invalid, also when a range overlaps with another. E1113

If the new value causes 'fillchars' or 'listchars' to become invalid it is rejected and an error is given.

To clear the overrides pass an empty {list}:

```
call setcellwidths([])
```

You can use the script $VIMRUNTIME/scripts/emoji_list.lua to see the effect for known emoji characters. Move the cursor through the text to check if the cell widths of your terminal match with what Vim knows about each emoji. If it doesn't look right you need to adjust the {list} argument.

getcellwidths() can be used to get the overrides later.

Parameters:
- {list} (any[])

Return:
(any)

setcharpos({expr}, {list}) setcharpos()

Lua: Prefer nvim_win_set_cursor() or nvim_buf_set_mark()

with vim.str_byteindex(); special positions, off, and
curswant differ.

Same as setpos() but uses the specified column number as the
character index instead of the byte index in the line.

Example:
With the text "□□□□" in line 8:

```
call setcharpos('.', [0, 8, 4, 0])
```

positions the cursor on the fourth character '□'.

```
call setpos('.', [0, 8, 4, 0])
```

positions the cursor on the second character '□'.

Parameters:
- {expr} (string)

- {list} (integer[])

Return:
(any)

setcharsearch({dict}) setcharsearch()

Set the current character search information to {dict},
which contains one or more of the following entries:

char	character which will be used for a subsequent , or ; command; an empty string clears the character search
forward	direction of character search; 1 for forward, 0 for backward
until	type of character search; 1 for a t or T character search, 0 for an f or F character search

This can be useful to save/restore a user's character search
from a script:

```
let prevsearch = getcharsearch()
" Perform a command which clobbers user's search
call setcharsearch(prevsearch)
```

Also see getcharsearch().

Parameters:
- {dict} ({ char?: string, forward?: 1|0, until?: 1|0 })

Return:
(any)

setcmdline({str} [, {pos}]) setcmdline()

Set the command line to {str} and set the cursor position to
{pos}.
If {pos} is omitted, the cursor is positioned after the text.
Returns 0 when successful, 1 when not editing the command
line.

Parameters:
- {str} (string)

- {pos} (integer?)

Return:
(integer)

setcmdpos({pos}) setcmdpos()

Set the cursor position in the command line to byte position
{pos}. The first position is 1.
Use getcmdpos() to obtain the current position.
Only works while editing the command line, thus you must use
c_CTRL-_e, c_CTRL-R_= or c_CTRL-R_CTRL-R with '='. For
c_CTRL-_e and c_CTRL-R_CTRL-R with '=' the position is
set after the command line is set to the expression. For
c_CTRL-R_= it is set after evaluating the expression but
before inserting the resulting text.
When the number is too big the cursor is put at the end of the
line. A number smaller than one has undefined results.
Returns 0 when successful, 1 when not editing the command
line.

Parameters:
- {pos} (integer)

Return:
(any)

setcursorcharpos({lnum}, {col} [, {off}]) setcursorcharpos()

Lua: Prefer nvim_win_set_cursor() with vim.str_byteindex(); off, return values, and special line forms differ.

setcursorcharpos({list})
Lua: Prefer nvim_win_set_cursor() with vim.str_byteindex(); list shape, off, and return values differ.

Same as cursor() but uses the specified column number as the character index instead of the byte index in the line.

Example:
With the text "□□□□" in line 4:

```
call setcursorcharpos(4, 3)
```

positions the cursor on the third character '□'.

```
call cursor(4, 3)
```

positions the cursor on the first character '□'.

Returns 0 when the position could be set, -1 otherwise.

Parameters:
- {list} (integer[])

Return:
(any)

setenv({name}, {val}) setenv()

Lua: Prefer vim.env.

Set environment variable {name} to {val}. Example:

```
call setenv('HOME', '/home/myhome')
```

When {val} is v:null the environment variable is deleted.
See also expr-env.

Parameters:
- {name} (string)
- {val} (string)

Return:
(any)

setfperm({fname}, {mode}) setfperm() chmod
Lua: Prefer uv.fs_chmod().

Set the file permissions for {fname} to {mode}.
{mode} must be a string with 9 characters. It is of the form "rwxrwxrwx", where each group of "rwx" flags represent, in turn, the permissions of the owner of the file, the group the file belongs to, and other users. A '-' character means the permission is off, any other character means on. Multi-byte characters are not supported.

For example "rw-r-----" means read-write for the user, readable by the group, not accessible by others. "xx-x-----" would do the same thing.

Returns non-zero for success, zero for failure.

To read permissions see getfperm().

Parameters:
- {fname} (string)
- {mode} (string)

Return:
(any)

setline({lnum}, {text}) setline()

Lua: Prefer nvim_set_current_line() or nvim_buf_set_lines().

Set line {lnum} of the current buffer to {text}. To insert lines use append(). To set lines in another buffer use setbufline().

{lnum} is used like with getline().
When {lnum} is just below the last line the {text} will be added below the last line.

{text} can be any type or a List of any type, each item is converted to a String. When {text} is an empty List then nothing is changed and FALSE is returned.

If this succeeds, FALSE is returned. If this fails (most likely because {lnum} is invalid) TRUE is returned.

Example:

```
call setline(5, strftime("%c"))
```

When {text} is a List then line {lnum} and following lines will be set to the items in the list. Example:

```
call setline(5, ['aaa', 'bbb', 'ccc'])
```

This is equivalent to:

```
for [n, l] in [[5, 'aaa'], [6, 'bbb'], [7, 'ccc']]
  call setline(n, l)
endfor
```

Note: The '[and '] marks are not set.

Parameters:

- {lnum} (integer|string)
- {text} (any)

Return:

(any)

setloclist({nr}, {list} [, {action} [, {what}]]) setloclist()

Create or replace or add to the location list for window {nr} (window-number or window-ID).
When {nr} is zero the current window is used.

For a location list window, the displayed location list is modified. For an invalid {nr}, -1 is returned.
Otherwise, same as setqflist().
Also see location-list.

For {action} see setqflist-action.

If the optional {what} dictionary argument is supplied, then only the items listed in {what} are set. Refer to setqflist() for the list of supported keys in {what}.

Parameters:

- {nr} (integer)
- {list} (vim.quickfix.entry[])
- {action} (string?)
- {what} (vim.fn.setqflist.what?)

Return:

(any)

setmatches({list} [, {win}]) setmatches()

Restores a list of matches saved by getmatches() for the current window. Returns 0 if successful, otherwise -1. All current matches are cleared before the list is restored. See example for getmatches().
{win} (if given) is a window-number or window-ID.

Parameters:

- {list} (vim.fn.getmatches.ret.item[])
- {win} (integer?)

Return:

(any)

setpos({expr}, {list}) setpos()

Lua: Prefer nvim_win_set_cursor() for "." or nvim_buf_set_mark() for marks; special positions, off, and curswant differ.

Set the position for String {expr}. Possible values:

```
.       the cursor
'x      mark x
```

{list} must be a List with four or five numbers:
    [bufnum, lnum, col, off]
    [bufnum, lnum, col, off, curswant]

"bufnum" is the buffer number. Zero can be used for the current buffer. When setting an uppercase mark "bufnum" is used for the mark position. For other marks it specifies the buffer to set the mark in. You can use the bufnr() function to turn a file name into a buffer number.
For setting the cursor and the ' mark "bufnum" is ignored, since these are associated with a window, not a buffer.
Does not change the jumplist.

"lnum" and "col" are the position in the buffer. The first column is 1. Use a zero "lnum" to delete a mark. If "col" is smaller than 1 then 1 is used. To use the character count instead of the byte count, use setcharpos().

The "off" number is only used when 'virtualedit' is set. Then it is the offset in screen columns from the start of the character. E.g., a position within a <Tab> or after the last character.

The "curswant" number is only used when setting the cursor position. It sets the preferred column for when moving the cursor vertically. When the "curswant" number is missing the preferred column is not set. When it is present and setting a mark position it is not used.

Note that for '< and '> changing the line number may result in the marks to be effectively swapped, so that '< is always before '>.

The visual marks '< and '> refer to the beginning and end of the visual selection relative to the cursor position.
Note that this differs from getpos(), where they are relative to the buffer.

Returns 0 when the position could be set, -1 otherwise.
An error message is given if {expr} is invalid.

Also see setcharpos(), getpos() and getcurpos().

This does not restore the preferred column for moving vertically; if you set the cursor position with this, j and k motions will jump to previous columns! Use cursor() to also set the preferred column. Also see the "curswant" key in winrestview().

Parameters:

- {expr} (string)
- {list} (integer[])

Return:

(any)

setqflist({list} [, {action} [, {what}]]) setqflist()

Create or replace or add to the quickfix list.

If the optional {what} dictionary argument is supplied, then only the items listed in {what} are set. The first {list} argument is ignored. See below for the supported items in {what}.

setqflist-what

When {what} is not present, the items in {list} are used. Each item must be a dictionary. Non-dictionary items in {list} are ignored. Each dictionary item can contain the following entries:

bufnr	buffer number; must be the number of a valid buffer
filename	name of a file; only used when "bufnr" is not present or it is invalid.
module	name of a module; if given it will be used in quickfix error window instead of the filename.
lnum	line number in the file
end_lnum	end of lines, if the item spans multiple lines
pattern	search pattern used to locate the error
col	column number
vcol	when non-zero: "col" is visual column when zero: "col" is byte index
end_col	end column, if the item spans multiple columns
nr	error number
text	description of the error
type	single-character error type, 'E', 'W', etc.
valid	recognized error message
user_data	custom data associated with the item, can be any type.

The "col", "vcol", "nr", "type" and "text" entries are optional. Either "lnum" or "pattern" entry can be used to

locate a matching error line.
If the "filename" and "bufnr" entries are not present or
neither the "lnum" or "pattern" entries are present, then the
item will not be handled as an error line.
If both "pattern" and "lnum" are present then "pattern" will
be used.
If the "valid" entry is not supplied, then the valid flag is
set when "bufnr" is a valid buffer or "filename" exists.
If you supply an empty {list}, the quickfix list will be
cleared.
Note that the list is not exactly the same as what
getqflist() returns.

{action} values: setqflist-action E927

'a' The items from {list} are added to the existing quickfix list. If there is no existing list, then a new list is created.

'r' The items from the current quickfix list are replaced with the items from {list}. This can also be used to clear the list:

```
call setqflist([], 'r')
```

'u' Like 'r', but tries to preserve the current selection in the quickfix list. The entry nearest to the previously selected one becomes the current entry. Proximity is determined by comparing the file, then the line number and then the column number.

'f' All the quickfix lists in the quickfix stack are freed.

If {action} is not present or is set to ' ', then a new list is created. The new quickfix list is added after the current quickfix list in the stack and all the following lists are freed. To add a new quickfix list at the end of the stack, set "nr" in {what} to "$".

The following items can be specified in dictionary {what}:

- context: quickfix list context. See quickfix-context
- efm: errorformat to use when parsing text from "lines". If this is not present, then the 'errorformat' option value is used. See quickfix-parse
- id: quickfix list identifier quickfix-ID
- idx: index of the current entry in the quickfix list specified by "id" or "nr". If set to '$', then the last entry in the list is set as the current entry. See quickfix-index
- items: list of quickfix entries. Same as the {list} argument.
- lines: use 'errorformat' to parse a list of lines and add the resulting entries to the quickfix list {nr} or {id}. Only a List value is supported. See quickfix-parse
- nr: list number in the quickfix stack; zero means the current quickfix list and "$" means the last quickfix list.
- quickfixtextfunc: function to get the text to display in the quickfix window. The value can be the name of a function or a funcref or a lambda. Refer to quickfix-window-function for an explanation of how to write the function and an example.
- title: quickfix list title text. See quickfix-title

Unsupported keys in {what} are ignored.
If the "nr" item is not present, then the current quickfix list is modified. When creating a new quickfix list, "nr" can be set to a value one greater than the quickfix stack size.
When modifying a quickfix list, to guarantee that the correct list is modified, "id" should be used instead of "nr" to specify the list.

Examples (See also setqflist-examples):

```
call setqflist([], 'r', {'title': 'My search'})
call setqflist([], 'r', {'nr': 2, 'title': 'Errors'})
call setqflist([], 'a', {'id':qfid, 'lines':["F1:10:L10"]})
```

Returns zero for success, -1 for failure.

This function can be used to create a quickfix list independent of the 'errorformat' setting. Use a command like :cc 1 to jump to the first position.

Parameters:

- {list} (vim.quickfix.entry[])
- {action} (string?)
- {what} (vim.fn.setqflist.what?)

Return:
(integer)

setreg({regname}, {value} [, {options}]) setreg()

Set the register {regname} to {value}.
If {regname} is "" or "@", the unnamed register '"' is used.
The {regname} argument is a string.

{value} may be any value returned by getreg() or
getreginfo(), including a List or Dict.
If {options} contains "a" or {regname} is upper case,
then the value is appended.

{options} can also contain a register type specification:
    "c" or "v"   charwise mode
    "l" or "V"   linewise mode
    "b" or "<CTRL-V>" blockwise-visual mode
If a number immediately follows "b" or "<CTRL-V>" then this is
used as the width of the selection - if it is not specified
then the width of the block is set to the number of characters
in the longest line (counting a <Tab> as 1 character).
If {options} contains "u" or '"', then the unnamed register is
set to point to register {regname}.

If {options} contains no register settings, then the default
is to use character mode unless {value} ends in a <NL> for
string {value} and linewise mode for list {value}. Blockwise
mode is never selected automatically.
Returns zero for success, non-zero for failure.

E883

Note: you may not use List containing more than one item to
set search and expression registers. Lists containing
no items act like empty strings.

Examples:

```
call setreg(v:register, @*)
call setreg('*', @%, 'ac')
call setreg('a', "1\n2\n3", 'b5')
call setreg('"', { 'points_to': 'a'})
```

This example shows using the functions to save and restore a
register:

```
let var_a = getreginfo()
call setreg('a', var_a)
```

or:

```
let var_a = getreg('a', 1, 1)
let var_amode = getregtype('a')
" ....
call setreg('a', var_a, var_amode)
```

Note: you may not reliably restore register value
without using the third argument to getreg() as without it
newlines are represented as newlines AND Nul bytes are
represented as newlines as well, see NL-used-for-Nul.

You can also change the type of a register by appending
nothing:

```
call setreg('a', '', 'al')
```

Parameters:
- {regname} (string)
- {value} (any)
- {options} (string?)

Return:
(any)

settabvar({tabnr}, {varname}, {val}) settabvar()

Lua: Prefer nvim_tabpage_set_var() or vim.t.

Set tab-local variable {varname} to {val} in tabpage {tabnr}.
t:var
The {varname} argument is a string.
Note that the variable name without "t:" must be used.
Tabs are numbered starting with one.
This function is not available in the sandbox.

Parameters:
- {tabnr} (integer)

- {varname} (string)

- {val} (any)

Return:
(any)

settabwinvar({tabnr}, {winnr}, {varname}, {val}) settabwinvar()

Lua: Prefer nvim_win_set_var() or vim.w after resolving {tabnr} and {winnr} to a winid; option names use nvim_set_option_value().

Set option or local variable {varname} in window {winnr} to {val}.
Tabs are numbered starting with one. For the current tabpage use setwinvar().
{winnr} is a window-number or window-ID.
When {winnr} is zero the current window is used.
This also works for a global or local buffer option, but it doesn't work for a global or local buffer variable.
For a local buffer option the global value is unchanged.
Note that the variable name without "w:" must be used.
Examples:

```
call settabwinvar(1, 1, "&list", 0)
call settabwinvar(3, 2, "myvar", "foobar")
```

This function is not available in the sandbox.

Parameters:
- {tabnr} (integer)

- {winnr} (integer)

- {varname} (string)

- {val} (any)

Return:
(any)

settagstack({nr}, {dict} [, {action}]) settagstack()

Modify the tag stack of the window {nr} (window-number or window-ID) using {dict}.

For a list of supported items in {dict}, refer to gettagstack(). "curidx" takes effect before changing the tag stack.

E962

How the tag stack is modified depends on the {action} argument:
- If {action} is not present or is set to 'r', then the tag stack is replaced.
- If {action} is set to 'a', then new entries from {dict} are pushed (added) onto the tag stack.
- If {action} is set to 't', then all the entries from the current entry in the tag stack or "curidx" in {dict} are removed and then new entries are pushed to the stack.

The current index is set to one after the length of the tag stack after the modification.

Returns zero for success, -1 for failure.

Examples (for more examples see tagstack-examples):
Empty the tag stack of window 3:

```
call settagstack(3, {'items' : []})
```

Save and restore the tag stack:

```
let stack = gettagstack(1003)
" do something else
call settagstack(1003, stack)
unlet stack
```

Parameters:

- {nr} (integer)
- {dict} (any)
- {action} (string?)

Return:
(any)

setwinvar({nr}, {varname}, {val}) setwinvar()

Lua: Prefer nvim_win_set_var() or vim.w after resolving
{nr} to a winid; option names use nvim_set_option_value().

Like settabwinvar() for the current tabpage.
Examples:

```
call setwinvar(1, "&list", 0)
call setwinvar(2, "myvar", "foobar")
```

Parameters:
- {nr} (integer)
- {varname} (string)
- {val} (any)

Return:
(any)

sha256({expr}) sha256()

Returns a String with 64 hex characters, which is the SHA256
checksum of {expr}.
{expr} is a String or a Blob.

Parameters:
- {expr} (string)

Return:
(string)

shellescape({string} [, {special}]) shellescape()

Lua: Prefer vim.system() with list args to avoid shell
escaping; :! and {special} semantics differ.

Escape {string} for use as a shell command argument.

On Windows when 'shellslash' is not set, encloses {string} in
double-quotes and doubles all double-quotes within {string}.
Otherwise encloses {string} in single-quotes and replaces all
"'" with "'\''".

The {special} argument adds additional escaping of keywords
used in Vim commands. If it is a non-zero-arg:
- Special items such as "!", "%", "#" and "<cword>" (as listed
  in expand()) will be preceded by a backslash.
  The backslash will be removed again by the :! command.
- The <NL> character is escaped.

If 'shell' contains "csh" in the tail:
- The "!" character will be escaped. This is because csh and
  tcsh use "!" for history replacement even in single-quotes.
- The <NL> character is escaped (twice if {special} is
  a non-zero-arg).

If 'shell' contains "fish" in the tail, the "\" character will
be escaped because in fish it is used as an escape character
inside single quotes.

Example of use with a :! command:

```
exe '!dir ' .. shellescape(expand('<cfile>'), 1)
```

This results in a directory listing for the file under the
cursor. Example of use with system():

```
call system("chmod +w -- " .. shellescape(expand("%")))
```

See also ::S.

Parameters:

- {string} (string)

- {special} (boolean?)

Return:

(string)

shiftwidth([{col}]) shiftwidth()

Lua: Prefer vim.bo for simple cases:
vim.bo.shiftwidth > 0 and vim.bo.shiftwidth or vim.bo.tabstop;
{col} and 'vartabstop' semantics differ.

Returns the effective value of 'shiftwidth'. This is the
'shiftwidth' value unless it is zero, in which case it is the
'tabstop' value. To be backwards compatible in indent
plugins, use this:

```
if exists('*shiftwidth')
  func s:sw()
    return shiftwidth()
  endfunc
else
  func s:sw()
    return &sw
  endfunc
endif
```

And then use s:sw() instead of &sw.

for which to return the 'shiftwidth' value. This matters for
the 'vartabstop' feature. If the 'vartabstop' setting is
enabled and no {col} argument is given, column 1 will be
assumed.

Parameters:

- {col} (integer?)

Return:

(integer)

sign_define({name} [, {dict}]) sign_define()

sign_define({list})

Define a new sign named {name} or modify the attributes of an
existing sign. This is similar to the :sign-define command.

Prefix {name} with a unique text to avoid name collisions.
There is no {group} like with placing signs.

The {name} can be a String or a Number. The optional {dict}
argument specifies the sign attributes. The following values
are supported:

icon	full path to the bitmap file for the sign.
linehl	highlight group used for the whole line the sign is placed in.
priority	default priority value of the sign
numhl	highlight group used for the line number where the sign is placed.
text	text that is displayed when there is no icon or the GUI is not being used.
texthl	highlight group used for the text item
culhl	highlight group used for the text item when the cursor is on the same line as the sign and 'cursorline' is enabled.

If the sign named {name} already exists, then the attributes
of the sign are updated.

The one argument {list} can be used to define a list of signs.
Each list item is a dictionary with the above items in {dict}
and a "name" item for the sign name.

Returns 0 on success and -1 on failure. When the one argument
{list} is used, then returns a List of values one for each
defined sign.

Examples:

```
call sign_define("mySign", {
        \ "text" : "=>",
        \ "texthl" : "Error",
        \ "linehl" : "Search"})
call sign_define([
        \ {'name' : 'sign1',
        \  'text' : '=>'},
        \ {'name' : 'sign2',
        \  'text' : '!!'}
        \ ])
```

Parameters:
- {list} (vim.fn.sign_define.dict[])

Return:
((0|-1)[])

sign_getdefined([{name}]) sign_getdefined()

Get a list of defined signs and their attributes.
This is similar to the :sign-list command.

If the {name} is not supplied, then a list of all the defined signs is returned. Otherwise the attribute of the specified sign is returned.

Each list item in the returned value is a dictionary with the following entries:

icon	full path to the bitmap file of the sign
linehl	highlight group used for the whole line the sign is placed in; not present if not set.
name	name of the sign
priority	default priority value of the sign
numhl	highlight group used for the line number where the sign is placed; not present if not set.
text	text that is displayed when there is no icon or the GUI is not being used.
texthl	highlight group used for the text item; not present if not set.
culhl	highlight group used for the text item when the cursor is on the same line as the sign and 'cursorline' is enabled; not present if not set.

Returns an empty List if there are no signs and when {name} is not found.

Examples:

```
" Get a list of all the defined signs
echo sign_getdefined()

" Get the attribute of the sign named mySign
echo sign_getdefined("mySign")
```

Parameters:
- {name} (string?)

Return:
(vim.fn.sign_getdefined.ret.item[])

sign_getplaced([{buf} [, {dict}]]) sign_getplaced()

Return a list of signs placed in a buffer or all the buffers.
This is similar to the :sign-place-list command.

If the optional buffer name {buf} is specified, then only the list of signs placed in that buffer is returned. For the use of {buf}, see bufname(). The optional {dict} can contain the following entries:

group	select only signs in this group
id	select sign with this identifier
lnum	select signs placed in this line. For the use of {lnum}, see line().

If {group} is "*", then signs in all the groups including the global group are returned. If {group} is not supplied or is an empty string, then only signs in the global group are returned. If no arguments are supplied, then signs in the global group placed in all the buffers are returned.
See sign-group.

Each list item in the returned value is a dictionary with the following entries:

bufnr	number of the buffer with the sign
signs	list of signs placed in {bufnr}. Each list item is a dictionary with the below listed entries

The dictionary for each sign contains the following entries:

group	sign group. Set to '' for the global group.
id	identifier of the sign
lnum	line number where the sign is placed
name	name of the defined sign
priority	sign priority

The returned signs in a buffer are ordered by their line number and priority.

Returns an empty list on failure or if there are no placed signs.

Examples:

```
" Get a List of signs placed in eval.c in the
" global group
echo sign_getplaced("eval.c")

" Get a List of signs in group 'g1' placed in eval.c
echo sign_getplaced("eval.c", {'group' : 'g1'})

" Get a List of signs placed at line 10 in eval.c
echo sign_getplaced("eval.c", {'lnum' : 10})

" Get sign with identifier 10 placed in a.py
echo sign_getplaced("a.py", {'id' : 10})

" Get sign with id 20 in group 'g1' placed in a.py
echo sign_getplaced("a.py", {'group' : 'g1',
                             \  'id' : 20})

" Get a List of all the placed signs
echo sign_getplaced()
```

Parameters:

- {buf} (integer|string?)
- {dict} (vim.fn.sign_getplaced.dict?)

Return:

(vim.fn.sign_getplaced.ret.item[])

sign_jump({id}, {group}, {buf}) sign_jump()

Open the buffer {buf} or jump to the window that contains {buf} and position the cursor at sign {id} in group {group}. This is similar to the :sign-jump command.

If {group} is an empty string, then the global group is used. For the use of {buf}, see bufname().

Returns the line number of the sign. Returns -1 if the arguments are invalid.

Example:

```
" Jump to sign 10 in the current buffer
call sign_jump(10, '', '')
```

Parameters:

- {id} (integer)
- {group} (string)
- {buf} (integer|string)

Return:

(integer)

sign_place({id}, {group}, {name}, {buf} [, {dict}]) sign_place()

Place the sign defined as {name} at line {lnum} in file or buffer {buf} and assign {id} and {group} to sign. This is similar to the :sign-place command.

If the sign identifier {id} is zero, then a new identifier is allocated. Otherwise the specified number is used. {group} is the sign group name. To use the global sign group, use an empty string. {group} functions as a namespace for {id}, thus two groups can use the same IDs. Refer to sign-identifier and sign-group for more information.

{name} refers to a defined sign.
{buf} refers to a buffer name or number. For the accepted values, see bufname().

The optional {dict} argument supports the following entries:

lnum	line number in the file or buffer {buf} where the sign is to be placed. For the accepted values, see line().
priority	priority of the sign. See sign-priority for more information.

If the optional {dict} is not specified, then it modifies the placed sign {id} in group {group} to use the defined sign {name}.

Returns the sign identifier on success and -1 on failure.

Examples:

```
" Place a sign named sign1 with id 5 at line 20 in
" buffer json.c
call sign_place(5, '', 'sign1', 'json.c',
                               \ {'lnum' : 20})

" Updates sign 5 in buffer json.c to use sign2
call sign_place(5, '', 'sign2', 'json.c')

" Place a sign named sign3 at line 30 in
" buffer json.c with a new identifier
let id = sign_place(0, '', 'sign3', 'json.c',
                               \ {'lnum' : 30})

" Place a sign named sign4 with id 10 in group 'g3'
" at line 40 in buffer json.c with priority 90
call sign_place(10, 'g3', 'sign4', 'json.c',
                \ {'lnum' : 40, 'priority' : 90})
```

Parameters:

- {id} (integer)
- {group} (string)
- {name} (string)
- {buf} (integer|string)
- {dict} (vim.fn.sign_place.dict?)

Return:

(integer)

sign_placelist({list}) sign_placelist()

Place one or more signs. This is similar to the sign_place() function. The {list} argument specifies the List of signs to place. Each list item is a dict with the following sign attributes:

buffer	Buffer name or number. For the accepted values, see bufname().
group	Sign group. {group} functions as a namespace for {id}, thus two groups can use the same IDs. If not specified or set to an empty string, then the global group is used. See sign-group for more information.
id	Sign identifier. If not specified or zero, then a new unique identifier is allocated. Otherwise the specified number is used. See sign-identifier for more information.
lnum	Line number in the buffer where the sign is to be placed. For the accepted values, see line().
name	Name of the sign to place. See sign_define() for more information.
priority	Priority of the sign. When multiple signs are placed on a line, the sign with the highest priority is used. If not specified, the default value of 10 is used, unless specified otherwise by the sign definition. See sign-priority for more information.

If {id} refers to an existing sign, then the existing sign is modified to use the specified {name} and/or {priority}.

Returns a List of sign identifiers. If failed to place a sign, the corresponding list item is set to -1.

Examples:

```
" Place sign s1 with id 5 at line 20 and id 10 at line
" 30 in buffer a.c
let [n1, n2] = sign_placelist([
        \ {'id' : 5,
        \  'name' : 's1',
        \  'buffer' : 'a.c',
        \  'lnum' : 20},
        \ {'id' : 10,
        \  'name' : 's1',
        \  'buffer' : 'a.c',
        \  'lnum' : 30}
        \ ])

" Place sign s1 in buffer a.c at line 40 and 50
" with auto-generated identifiers
let [n1, n2] = sign_placelist([
        \ {'name' : 's1',
        \  'buffer' : 'a.c',
        \  'lnum' : 40},
        \ {'name' : 's1',
        \  'buffer' : 'a.c',
        \  'lnum' : 50}
        \ ])
```

Parameters:
- {list} (vim.fn.sign_placelist.list.item[])

Return:
(integer[])

sign_undefine([{name}]) sign_undefine()

sign_undefine({list})
Deletes a previously defined sign {name}. This is similar to the :sign-undefine command. If {name} is not supplied, then deletes all the defined signs.

The one argument {list} can be used to undefine a list of signs. Each list item is the name of a sign.

Returns 0 on success and -1 on failure. For the one argument {list} call, returns a list of values one for each undefined sign.

Examples:

```
" Delete a sign named mySign
call sign_undefine("mySign")

" Delete signs 'sign1' and 'sign2'
call sign_undefine(["sign1", "sign2"])

" Delete all the signs
call sign_undefine()
```

Parameters:
- {list} (string[]?)

Return:
(integer[])

sign_unplace({group} [, {dict}]) sign_unplace()

Remove a previously placed sign in one or more buffers. This is similar to the :sign-unplace command.

{group} is the sign group name. To use the global sign group, use an empty string. If {group} is set to "*", then all the groups including the global group are used.
The signs in {group} are selected based on the entries in {dict}. The following optional entries in {dict} are supported:
        buffer buffer name or number. See bufname().
        id   sign identifier
If {dict} is not supplied, then all the signs in {group} are removed.

Returns 0 on success and -1 on failure.

Examples:

```
" Remove sign 10 from buffer a.vim
call sign_unplace('', {'buffer' : "a.vim", 'id' : 10})

" Remove sign 20 in group 'g1' from buffer 3
call sign_unplace('g1', {'buffer' : 3, 'id' : 20})

" Remove all the signs in group 'g2' from buffer 10
call sign_unplace('g2', {'buffer' : 10})

" Remove sign 30 in group 'g3' from all the buffers
call sign_unplace('g3', {'id' : 30})

" Remove all the signs placed in buffer 5
call sign_unplace('*', {'buffer' : 5})

" Remove the signs in group 'g4' from all the buffers
call sign_unplace('g4')

" Remove sign 40 from all the buffers
call sign_unplace('*', {'id' : 40})

" Remove all the placed signs from all the buffers
call sign_unplace('*')
```

Parameters:
- {group} (string)

- {dict} (vim.fn.sign_unplace.dict?)

Return:
(0|-1)

sign_unplacelist({list}) sign_unplacelist()

Remove previously placed signs from one or more buffers. This is similar to the sign_unplace() function.

The {list} argument specifies the List of signs to remove. Each list item is a dict with the following sign attributes:

buffer	buffer name or number. For the accepted values, see bufname(). If not specified, then the specified sign is removed from all the buffers.
group	sign group name. If not specified or set to an empty string, then the global sign group is used. If set to "*", then all the groups including the global group are used.
id	sign identifier. If not specified, then all the signs in the specified group are removed.

Returns a List where an entry is set to 0 if the corresponding sign was successfully removed or -1 on failure.

Example:

```
" Remove sign with id 10 from buffer a.vim and sign
" with id 20 from buffer b.vim
call sign_unplacelist([
        \ {'id' : 10, 'buffer' : "a.vim"},
        \ {'id' : 20, 'buffer' : 'b.vim'},
        \ ])
```

Parameters:
- {list} (vim.fn.sign_unplacelist.list.item)

Return:
((0|-1)[])

simplify({filename}) simplify()

Lua: Prefer vim.fs.normalize(); ~/$VAR expansion and leading ./ or trailing / handling differ.

Simplify the file name as much as possible without changing the meaning. Shortcuts (on MS-Windows) or symbolic links (on Unix) are not resolved. If the first path component in {filename} designates the current directory, this will be valid for the result as well. A trailing path separator is not removed either. On Unix "//path" is unchanged, but "///path" is simplified to "/path" (this follows the Posix standard).
Example:

```
simplify("./dir/.././/file/") == "./file/"
```

Note: The combination "dir/.." is only removed if "dir" is a searchable directory or does not exist. On Unix, it is also removed when "dir" is a symbolic link within the same

directory. In order to resolve all the involved symbolic
links before simplifying the path name, use resolve().

Parameters:
- {filename} (string)

Return:
(string)

sin({expr}) sin()

Lua: Prefer math.sin().

Return the sine of {expr}, measured in radians, as a Float.
{expr} must evaluate to a Float or a Number.
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo sin(100)
```

-0.506366

```
echo sin(-4.01)
```

0.763301

Parameters:
- {expr} (number)

Return:
(number)

sinh({expr}) sinh()

Lua: Prefer math.sinh().

Return the hyperbolic sine of {expr} as a Float in the range
[-inf, inf].
{expr} must evaluate to a Float or a Number.
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo sinh(0.5)
```

0.521095

```
echo sinh(-0.9)
```

-1.026517

Parameters:
- {expr} (number)

Return:
(any)

slice({expr}, {start} [, {end}]) slice()

Similar to using a slice "expr[start : end]", but "end" is
used exclusive. And for a string the indexes are used as
character indexes instead of byte indexes.
Also, composing characters are treated as a part of the
preceding base character.
When {end} is omitted the slice continues to the last item.
When {end} is -1 the last item is omitted.
Returns an empty value if {start} or {end} are invalid.

Parameters:
- {expr} (any)
- {start} (integer)
- {end} (integer?)

Return:
(any)

sockconnect({mode}, {address} [, {opts}]) sockconnect()

Connect a socket to an address. If {mode} is "pipe" then
{address} should be the path of a local domain socket (on
unix) or named pipe (on Windows). If {mode} is "tcp" then
{address} should be of the form "host:port" where the host
should be an ip address or host name, and port the port
number.

For "pipe" mode, see luv-pipe-handle. For "tcp" mode, see
luv-tcp-handle.

Returns a channel ID. Close the socket with chanclose(). Use chansend() to send data over a bytes socket, and rpcrequest() and rpcnotify() to communicate with a RPC socket.

{opts} is an optional dictionary with these keys:
  on_data : callback invoked when data was read from socket
  data_buffered : read socket data in channel-buffered mode.
  rpc   : If set, msgpack-rpc will be used to communicate
            over the socket.

Returns:

- The channel ID on success (greater than zero)
- 0 on invalid arguments or connection failure.

Parameters:

- {mode} (string)
- {address} (string)
- {opts} (table?)

Return:

(any)

sort({list} [, {how} [, {dict}]]) sort() E702

Lua: Prefer table.sort() with a custom comparator; it is not stable and Vim's i/`l`/`n`/`N`/`f` modes differ.

Sort the items in {list} in-place. Returns {list}.

If you want a list to remain unmodified make a copy first:

```
let sortedlist = sort(copy(mylist))
```

When {how} is omitted or is a string, then sort() uses the string representation of each item to sort on. Numbers sort after Strings, Lists after Numbers. For sorting text in the current buffer use :sort.

When {how} is given and it is 'i' then case is ignored. For backwards compatibility, the value one can be used to ignore case. Zero means to not ignore case.

When {how} is given and it is 'l' then the current collation locale is used for ordering. Implementation details: strcoll() is used to compare strings. See :language to check or set the collation locale. v:collate can also be used to check the current locale. Sorting using the locale typically ignores case. Example:

```
" ö is sorted similarly to o with English locale.
language collate en_US.UTF8
echo sort(['n', 'o', 'O', 'ö', 'p', 'z'], 'l')
```

['n', 'o', 'O', 'ö', 'p', 'z']

```
" ö is sorted after z with Swedish locale.
language collate sv_SE.UTF8
echo sort(['n', 'o', 'O', 'ö', 'p', 'z'], 'l')
```

['n', 'o', 'O', 'p', 'z', 'ö']

This does not work properly on Mac.

When {how} is given and it is 'n' then all items will be sorted numerical (Implementation detail: this uses the strtod() function to parse numbers, Strings, Lists, Dicts and Funcrefs will be considered as being 0).

When {how} is given and it is 'N' then all items will be sorted numerical. This is like 'n' but a string containing digits will be used as the number they represent.

When {how} is given and it is 'f' then all items will be sorted numerical. All values must be a Number or a Float.

When {how} is a Funcref or a function name, this function is called to compare items. The function is invoked with two items as argument and must return zero if they are equal, 1 or bigger if the first one sorts after the second one, -1 or smaller if the first one sorts before the second one.

{dict} is for functions with the "dict" attribute. It will be used to set the local variable "self". Dictionary-function

The sort is stable, items which compare equal (as number or as string) will keep their relative position. E.g., when sorting on numbers, text strings will sort next to each other, in the same order as they were originally.

Example:

```
func MyCompare(i1, i2)
   return a:i1 == a:i2 ? 0 : a:i1 > a:i2 ? 1 : -1
endfunc
eval mylist->sort("MyCompare")
```

A shorter compare version for this specific simple case, which ignores overflow:

```
func MyCompare(i1, i2)
   return a:i1 - a:i2
endfunc
```

For a simple expression you can use a lambda:

```
eval mylist->sort({i1, i2 -> i1 - i2})
```

Parameters:

- {list} (T[])
- {how} (string|function?)
- {dict} (any?)

Return:

(T[])

soundfold({word}) soundfold()

Return the sound-folded equivalent of {word}. Uses the first language in 'spelllang' for the current window that supports soundfolding. 'spell' must be set. When no sound folding is possible the {word} is returned unmodified.
This can be used for making spelling suggestions. Note that the method can be quite slow.

Parameters:

- {word} (string)

Return:

(string)

spellbadword([{sentence}]) spellbadword()

Lua: Prefer vim.spell.check() for string input; it returns all matches with positions, not just the first one or cursor-moving behavior.

Without argument: The result is the badly spelled word under or after the cursor. The cursor is moved to the start of the bad word. When no bad word is found in the cursor line the result is an empty string and the cursor doesn't move.

With argument: The result is the first word in {sentence} that is badly spelled. If there are no spelling mistakes the result is an empty string.

The return value is a list with two items:

- The badly spelled word or an empty string.
- The type of the spelling error:

```
"bad"           spelling mistake
"rare"          rare word
"local"         word only valid in another region
"caps"          word should start with Capital
```

Example:

```
echo spellbadword("the quik brown fox")
" ['quik', 'bad']
```

The spelling information for the current window and the value of 'spelllang' are used.

Parameters:

- {sentence} (string?)

Return:

(any)

spellsuggest({word} [, {max} [, {capital}]]) spellsuggest()

Return a List with spelling suggestions to replace {word}. When {max} is given up to this number of suggestions are returned. Otherwise up to 25 suggestions are returned.

When the {capital} argument is given and it's non-zero only suggestions with a leading capital will be given. Use this after a match with 'spellcapcheck'.

{word} can be a badly spelled word followed by other text. This allows for joining two words that were split. The suggestions also include the following text, thus you can replace a line.

{word} may also be a good word. Similar words will then be returned. {word} itself is not included in the suggestions, although it may appear capitalized.

The spelling information for the current window is used. The values of 'spelllang' and 'spellsuggest' are used.

Parameters:

- {word} (string)
- {max} (integer?)
- {capital} (boolean?)

Return:

(string[])

split({string} [, {pattern} [, {keepempty}]]) split()

Lua: Prefer vim.split().

Make a List out of {string}. When {pattern} is omitted or empty each white space separated sequence of characters becomes an item.
Otherwise the string is split where {pattern} matches, removing the matched characters. 'ignorecase' is not used here, add \c to ignore case. /\c
When the first or last item is empty it is omitted, unless the {keepempty} argument is given and it's non-zero.
Other empty items are kept when {pattern} matches at least one character or when {keepempty} is non-zero.
Example:

```
let words = split(getline('.'), '\W\+')
```

To split a string in individual characters:

```
for c in split(mystring, '\zs') | endfor
```

If you want to keep the separator you can also use '\zs' at the end of the pattern:

```
echo split('abc:def:ghi', ':\zs')
```

```
['abc:', 'def:', 'ghi']
```

Splitting a table where the first element can be empty:

```
let items = split(line, ':', 1)
```

The opposite function is join().

Parameters:

- {string} (string)
- {pattern} (string?)
- {keepempty} (boolean?)

Return:

(string[])

sqrt({expr}) sqrt()

Lua: Prefer math.sqrt().

Return the non-negative square root of Float {expr} as a Float.
{expr} must evaluate to a Float or a Number. When {expr} is negative the result is NaN (Not a Number). Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo sqrt(100)
```

```
10.0
```

```
echo sqrt(-4.01)
```

str2float("nan")
NaN may be different, it depends on system libraries.

Parameters:
- {expr} (number)

Return:
(any)

srand([{expr}]) srand()

Lua: Prefer math.randomseed().

Initialize seed used by rand():
- If {expr} is not given, seed values are initialized by

reading from /dev/urandom, if possible, or using time(NULL)
a.k.a. epoch time otherwise; this only has second accuracy.
- If {expr} is given it must be a Number. It is used to

initialize the seed values. This is useful for testing or
when a predictable sequence is intended.

Examples:

```
let seed = srand()
let seed = srand(userinput)
echo rand(seed)
```

Parameters:
- {expr} (number?)

Return:
(any)

state([{what}]) state()

Return a string which contains characters indicating the
current state. Mostly useful in callbacks that want to do
work that may not always be safe. Roughly this works like:
- callback uses state() to check if work is safe to do.

Yes: then do it right away.
No: add to work queue and add a SafeState autocommand.
- When SafeState is triggered and executes your autocommand,

check with state() if the work can be done now, and if yes
remove it from the queue and execute.
Remove the autocommand if the queue is now empty.
Also see mode().

When {what} is given only characters in this string will be
added. E.g, this checks if the screen has scrolled:

```
if state('s') == ''
   " screen has not scrolled
```

These characters indicate the state, generally indicating that
something is busy:
m halfway a mapping, :normal command, feedkeys() or
stuffed command
o operator pending, e.g. after d
a Insert mode autocomplete active
x executing an autocommand
S not triggering SafeState, e.g. after f or a count
c callback invoked, including timer (repeats for
recursiveness up to "ccc")
s screen has scrolled for messages

Attributes:
api-fast

Parameters:
- {what} (string?)

Return:
(any)

stdioopen({opts}) stdioopen()

With --headless this opens stdin and stdout as a channel.
May be called only once. See channel-stdio. stderr is not
handled by this function, see v:stderr.

Close the stdio handles with chanclose(). Use chansend()
to send data to stdout, and rpcrequest() and rpcnotify()
to communicate over RPC.

{opts} is a dictionary with these keys:
on_stdin : callback invoked when stdin is written to.
on_print : callback invoked when Nvim needs to print a
message, with the message (whose type is string)
as sole argument.
stdin_buffered : read stdin in channel-buffered mode.
rpc : If set, msgpack-rpc will be used to communicate
over stdio
Returns:

- channel-id on success (value is always 1)
- 0 on invalid arguments

Parameters:

- {opts} (table)

Return:

(any)

stdpath({what}) stdpath() E6100

Returns standard-path locations of various default files and directories. The locations are driven by base-directories which you can configure via $NVIM_APPNAME or the $XDG_… environment variables.

{what}	Type	Description
cache	String	Cache directory: arbitrary temporary storage for plugins, etc.
config	String	User configuration directory. init.vim is stored here.
config_dirs	List	Other configuration directories.
data	String	User data directory.
data_dirs	List	Other data directories.
log	String	Logs directory (for use by plugins too).
run	String	Run directory: temporary, local storage for sockets, named pipes, etc.
state	String	Session state: storage for backupdir, file drafts, shada, swap, undo, 'viewdir'.

Example:

```
echo stdpath("config")
```

Attributes:

api-fast

Parameters:

- {what}

('cache'|'config'|'config_dirs'|'data'|'data_dirs'|'log'|'run'|'state')

Return:

(string|string[])

str2float({string} [, {quoted}]) str2float()

Lua: Prefer tonumber() for simple numbers; quoted digits, hex forms, and trailing text differ.

Convert String {string} to a Float. This mostly works the same as when using a floating point number in an expression, see floating-point-format. But it's a bit more permissive. E.g., "1e40" is accepted, while in an expression you need to write "1.0e40". The hexadecimal form "0x123" is also accepted, but not others, like binary or octal.
When {quoted} is present and non-zero then embedded single quotes before the dot are ignored, thus "1'000.0" is a thousand.
Text after the number is silently ignored.
The decimal point is always '.', no matter what the locale is set to. A comma ends the number: "12,345.67" is converted to 12.0. You can strip out thousands separators with substitute():

```
let f = str2float(substitute(text, ',', '', 'g'))
```

Returns 0.0 if the conversion fails.

Parameters:

- {string} (string)
- {quoted} (boolean?)

Return:

(any)

str2list({string} [, {utf8}]) str2list()

Return a list containing the number values which represent
each character in String {string}. Examples:

```
echo str2list(" ")              " returns [32]
echo str2list("ABC")            " returns [65, 66, 67]
```

list2str() does the opposite.

UTF-8 encoding is always used, {utf8} option has no effect,
and exists only for backwards-compatibility.
With UTF-8 composing characters are handled properly:

```
echo str2list("á")              " returns [97, 769]
```

Attributes:
api-fast

Parameters:
- {string} (string)

- {utf8} (boolean?)

Return:
(any)

str2nr({string} [, {base}]) str2nr()

Lua: Prefer tonumber() for simple numbers; quoted digits,
base prefixes, and trailing text differ.

Convert string {string} to a number.
{base} is the conversion base, it can be 2, 8, 10 or 16.
When {quoted} is present and non-zero then embedded single
quotes are ignored, thus "1'000'000" is a million.

When {base} is omitted base 10 is used. This also means that
a leading zero doesn't cause octal conversion to be used, as
with the default String to Number conversion. Example:

```
let nr = str2nr('0123')
```

When {base} is 16 a leading "0x" or "0X" is ignored. With a
different base the result will be zero. Similarly, when
{base} is 8 a leading "0", "0o" or "0O" is ignored, and when
{base} is 2 a leading "0b" or "0B" is ignored.
Text after the number is silently ignored.

Returns 0 if {string} is empty or on error.

Parameters:
- {string} (string)

- {base} (integer?)

Return:
(any)

strcharlen({string}) strcharlen()

Returns the number of characters in String {string}, ignoring
composing characters. Returns 0 on error or empty {string}.

strchars() can count the number of characters, counting
composing characters separately.

Also see strlen(), strdisplaywidth() and strwidth().

Attributes:
api-fast

Parameters:
- {string} (string)

Return:
(integer)

strcharpart({src}, {start} [, {len} [, {skipcc}]]) strcharpart()

Like strpart() but using character index and length instead
of byte index and length.
When {skipcc} is omitted or zero, composing characters are
counted separately.
When {skipcc} set to 1, composing characters are treated as a
part of the preceding base character, similar to slice().
When a character index is used where a character does not
exist it is omitted and counted as one character. For
example:

```
echo strcharpart('abc', -1, 2)
```

results in 'a'.

Returns an empty string on error.

Attributes:

api-fast

Parameters:

- {src} (string)
- {start} (integer)
- {len} (integer?)
- {skipcc} (0|1|boolean?)

Return:

(string)

strchars({string} [, {skipcc}]) strchars()

Lua: Prefer vim.str_utfindex() for codepoint counts;
skipcc handling differs.

Returns the number of characters in String {string}.
When {skipcc} is omitted or zero, composing characters are
counted separately.
When {skipcc} set to 1, composing characters are ignored.
strcharlen() always does this.

Returns zero on error.

Also see strlen(), strdisplaywidth() and strwidth().

{skipcc} is only available after 7.4.755. For backward
compatibility, you can define a wrapper function:

```
if has("patch-7.4.755")
  function s:strchars(str, skipcc)
    return strchars(a:str, a:skipcc)
  endfunction
else
  function s:strchars(str, skipcc)
    if a:skipcc
      return strlen(substitute(a:str, ".", "x", "g"))
    else
      return strchars(a:str)
    endif
  endfunction
endif
```

Attributes:

api-fast

Parameters:

- {string} (string)
- {skipcc} (0|1|boolean?)

Return:

(integer)

strdisplaywidth({string} [, {col}]) strdisplaywidth()

Lua: Prefer nvim_strwidth() when {col} is 0; Tab expansion
and window-option semantics differ.

Returns the number of display cells String {string} occupies
on the screen when it starts at {col} (first column is zero).
Returns zero on error.

When {col} is omitted zero is used. Otherwise it is the screen
column where to start. This matters for Tab characters.
The option settings of the current window are used. This
matters for anything that's displayed differently, such as
'tabstop' and 'display'.

When {string} contains characters with East Asian Width Class
Ambiguous, this function's return value depends on
'ambiwidth'.

Also see strlen(), strwidth() and strchars().

Attributes:

api-fast

Parameters:

- {string} (string)
- {col} (integer?)

Return:
(integer)

strftime({format} [, {time}]) strftime()

Lua: Prefer os.date().

Formats a date and time String specified by {format}. The
given {time} is used, or the current time if no time is given.
The {format} depends on your system, this is not portable!
See the manual page of the C function strftime() for the
format. The maximum length of the result is 80 characters.
See also localtime(), getftime() and strptime().
The language can be changed with the :language command.
Examples:

```
echo strftime("%c")                " Sun Apr 27 11:49:23 1997
echo strftime("%Y %b %d %X")       " 1997 Apr 27 11:53:25
echo strftime("%y%m%d %T")         " 970427 11:53:55
echo strftime("%H:%M")             " 11:55
echo strftime("%c", getftime("file.c"))
                                 " Show mod time of file.c.
```

Parameters:
- {format} (string)

- {time} (number?)

Return:
(string)

strgetchar({str}, {index}) strgetchar()

Get a Number corresponding to the character at {index} in
{str}. This uses a zero-based character index, not a byte
index. Composing characters are considered separate
characters here. Use nr2char() to convert the Number to a
String.
Returns -1 if {index} is invalid.
Also see strcharpart() and strchars().

Attributes:
api-fast

Parameters:
- {str} (string)

- {index} (integer)

Return:
(integer)

stridx({haystack}, {needle} [, {start}]) stridx()

Lua: Prefer string.find() with plain = true and
start + 1, then subtract 1; missing matches return nil
instead of -1.

Returns the byte index of the first occurrence of {needle}
in {haystack}.
If {start} is specified, the search starts at index {start}.
This can be used to find a second match:

```
let colon1 = stridx(line, ":")
let colon2 = stridx(line, ":", colon1 + 1)
```

The search is done case-sensitive.
For pattern searches use match().
-1 is returned if the {needle} does not occur in {haystack}.
See also strridx().
Examples:

```
echo stridx("An Example", "Example")     " 3
echo stridx("Starting point", "Start")   " 0
echo stridx("Starting point", "start")   " -1
```

strstr() strchr()
stridx() works similar to the C function strstr(). When used
with a single character it works similar to strchr().

Attributes:
api-fast

Parameters:
- {haystack} (string)

- {needle} (string)

- {start} (integer?)

Return:
(integer)

`string({expr})` string()

Lua: Prefer vim.inspect() for tables, or tostring() for scalars.

Converts {expr} to a String. If {expr} is a Number, Float, String, Blob or a composition of them, the result can be parsed back with eval().

{expr} type	result
String	'string'
Number	123
Float	123.123456 or 1.123456e8 or str2float('inf')
Funcref	function('name')
Blob	0z00112233.44556677.8899
List	[item, item]
Dictionary	{key: value, key: value}

Note: in String values the ' character is doubled.
Also see strtrans().

Note: Output format is mostly compatible with YAML, except for infinite and NaN floating-point values representations which use str2float(). Strings are also dumped literally, only single quote is escaped, which does not allow using YAML for parsing back binary strings. eval() should always work for strings and floats though, and this is the only official method. Use msgpackdump() or json_encode() if you need to share data with other applications.

Parameters:
- {expr} (any)

Return:
(string)

`strlen({string})` strlen()

Lua: Prefer string.len().

Returns the length of String {string} in bytes.
If the argument is a Number it is first converted to a String.
For other types an error is given and zero is returned.
If you want to count the number of multibyte characters use strchars().
Also see len(), strdisplaywidth() and strwidth().

Attributes:
api-fast

Parameters:
- {string} (string)

Return:
(integer)

`strpart({src}, {start} [, {len} [, {chars}]])` strpart()

Lua: Prefer string.sub() for byte slices; negative start and the chars flag differ.

Gets a substring from {src}, starting from byte {start}, with byte length {len}. Returns empty string on error.

When {chars} is present and TRUE then {len} is the number of characters positions (composing characters are not counted separately, thus "1" means one base character and any following composing characters).

To count {start} as characters instead of bytes use strcharpart().

When bytes are selected which do not exist, this doesn't result in an error, the bytes are simply omitted.
If {len} is missing, the copy continues from {start} till the end of the {src}.

```
echo strpart("abcdefg", 3, 2)    " returns 'de'
echo strpart("abcdefg", -2, 4)   " returns 'ab'
echo strpart("abcdefg", 5, 4)    " returns 'fg'
echo strpart("abcdefg", 3)       " returns 'defg'
```

Note: To get the first character, {start} must be 0. For example, to get the character under the cursor:

```
strpart(getline("."), col(".") - 1, 1, v:true)
```

Attributes:
api-fast

Parameters:
- {src} (string)
- {start} (integer)
- {len} (integer?)
- {chars} (0|1?)

Return:
(string)

strptime({format}, {timestring}) strptime()

Parses a date/time string and returns a unix timestamp.
{timestring} must match the format specified in {format}.

The {format} depends on your system, this is not portable!
See the strptime() manpage for the format. Especially avoid
"%c". The value of $TZ also matters.

If the {timestring} cannot be parsed with {format} zero is
returned. If you do not know the format of {timestring} you
can try different {format} values until you get a non-zero
result.

See also strftime().
Examples:

```
echo strptime("%Y %b %d %X", "1997 Apr 27 11:49:23")
```

862156163

```
echo strftime("%c", strptime("%y%m%d %T", "970427 11:53:55"))
```

Sun Apr 27 11:53:55 1997

```
echo strftime("%c", strptime("%Y%m%d%H%M%S", "19970427115355") + 3600)
```

Sun Apr 27 12:53:55 1997

Parameters:
- {format} (string)
- {timestring} (string)

Return:
(integer)

strridx({haystack}, {needle} [, {start}]) strridx()

Lua: Prefer string.find() in a loop with plain = true,
then subtract 1; empty-needle and missing-match semantics
differ.

Returns the byte index of the last occurrence of {needle}
in {haystack}.
When {start} is specified, matches beyond this index are
ignored. This can be used to find a match before a previous
match:

```
let lastcomma = strridx(line, ",")
let comma2 = strridx(line, ",", lastcomma - 1)
```

The search is done case-sensitive.
For pattern searches use match().
-1 is returned if the {needle} does not occur in {haystack}.
If the {needle} is empty the length of {haystack} is returned.
See also stridx(). Examples:

```
echo strridx("an angry armadillo", "an")            3
```

strrchr()

When used with a single character it works similar to the C
function strrchr().

Parameters:
- {haystack} (string)
- {needle} (string)

- {start} (integer?)

Return:
(integer)

strtrans({string}) strtrans()

Translates all unprintable characters in {string} into
printable characters, like they are shown in a
window. Example:

```
echo strtrans(@a)
```

This displays a newline in register a as "^@" instead of
starting a new line.

Returns an empty string on error.

Attributes:
api-fast

Parameters:
- {string} (string)

Return:
(string)

strutf16len({string} [, {countcc}]) strutf16len()

Lua: Prefer vim.str_utfindex() with "utf-16";
countcc = false semantics differ.

Returns the number of UTF-16 code units in String {string}
(after converting it to UTF-16).

When {countcc} is TRUE, composing characters are counted
separately.
When {countcc} is omitted or FALSE, composing characters are
ignored.

Returns zero on error.

Also see strlen() and strcharlen().
Examples:

```
echo strutf16len('a')           " returns 1
echo strutf16len('©')           " returns 1
echo strutf16len('😊')          " returns 2
echo strutf16len('á')           " returns 1
echo strutf16len('á', v:true)   " returns 3
```

Parameters:
- {string} (string)

- {countcc} (0|1?)

Return:
(integer)

strwidth({string}) strwidth()

Lua: Prefer nvim_strwidth().

Returns the number of display cells String {string} occupies.
A Tab character is counted as one cell, alternatively use
strdisplaywidth().
When {string} contains characters with East Asian Width Class
Ambiguous, this function's return value depends on
'ambiwidth'.
Returns zero on error.
Also see strlen(), strdisplaywidth() and strchars().

Attributes:
api-fast

Parameters:
- {string} (string)

Return:
(integer)

submatch({nr} [, {list}]) submatch() E935
Only for an expression in a :substitute command or
substitute() function.
Returns the {nr}th submatch of the matched text. When {nr}
is 0 the whole matched text is returned.
Note that a NL in the string can stand for a line break of a
multi-line match or a NUL character in the text.
Also see sub-replace-expression.

If {list} is present and non-zero then submatch() returns
a list of strings, similar to getline() with two arguments.
NL characters in the text represent NUL characters in the
text.
Only returns more than one item for :substitute, inside
substitute() this list will always contain one or zero
items, since there are no real line breaks.

When substitute() is used recursively only the submatches in
the current (deepest) call can be obtained.

Returns an empty string or list on error.

Examples:

```
s/\d\+/\=submatch(0) + 1/
echo substitute(text, '\d\+', '\=submatch(0) + 1', '')
```

This finds the first number in the line and adds one to it.
A line break is included as a newline character.

Parameters:

- {nr} (integer)

- {list} (nil?)

Return:

(string)

substitute({string}, {pat}, {sub}, {flags}) substitute()

Lua: Prefer string.gsub() for common substitutions; Vim
patterns, flags, and replacement semantics differ.

Performs string substitution. Returns a copy of {string}
in which the first match of {pat} is replaced with {sub}.
When {flags} is "g", all matches of {pat} in {string} are
replaced. Otherwise {flags} should be "".

This works like the ":substitute" command (without any flags).
But the matching with {pat} is always done like the 'magic'
option is set and 'cpoptions' is empty (to make scripts
portable). 'ignorecase' is still relevant, use /\c or /\C
if you want to ignore or match case and ignore 'ignorecase'.
'smartcase' is not used. See string-match for how {pat} is
used.

A "~" in {sub} is not replaced with the previous {sub}.
Note that some codes in {sub} have a special meaning
sub-replace-special. For example, to replace something with
"\n" (two characters), use "\\\n" or '\\n'.

When {pat} does not match in {string}, {string} is returned
unmodified.

Example:

```
let &path = substitute(&path, ",\\=[^,]*$", "", "")
```

This removes the last component of the 'path' option.

```
echo substitute("testing", ".*", "\\U\\0", "")
```

results in "TESTING".

When {sub} starts with "\=", the remainder is interpreted as
an expression. See sub-replace-expression. Example:

```
echo substitute(s, '%\(\x\x\)',
   \ '\=nr2char("0x" .. submatch(1))', 'g')
```

When {sub} is a Funcref that function is called, with one
optional argument. Example:

```
echo substitute(s, '%\(\x\x\)', SubNr, 'g')
```

The optional argument is a list which contains the whole
matched string and up to nine submatches, like what
submatch() returns. Example:

```
echo substitute(s, '%\(\x\x\)', {m -> '0x' .. m[1]}, 'g')
```

Returns an empty string on error.

Parameters:

- {string} (string)

- {pat} (string)

- {sub} (string)

- {flags} (string)

Return:
(string)

swapfilelist() swapfilelist()

Returns a list of swap file names, like what "vim -r" shows.
See the -r command argument. The 'directory' option is used
for the directories to inspect. If you only want to get a
list of swap files in the current directory then temporarily
set 'directory' to a dot:

```
let save_dir = &directory
let &directory = '.'
let swapfiles = swapfilelist()
let &directory = save_dir
```

Return:
(string[])

swapinfo({fname}) swapinfo()

Returns information about the swapfile {fname} as a
dictionary. The available fields are:
version Vim version
user user name
host host name
fname original file name
pid PID of the Nvim process that created the swap
file, or zero if not running.
mtime last modification time in seconds
inode Optional: INODE number of the file
dirty 1 if file was modified, 0 if not
In case of failure an "error" item is added with the reason:
Cannot open file: file not found or in accessible
Cannot read file: cannot read first block
Not a swap file: does not contain correct block ID
Magic number mismatch: Info in first block is invalid

Parameters:
- {fname} (string)

Return:
(any)

swapname({buf}) swapname()

The result is the swap file path of the buffer {buf}.
For the use of {buf}, see bufname() above.
If buffer {buf} is the current buffer, the result is equal to
:swapname (unless there is no swap file).
If buffer {buf} has no swap file, returns an empty string.

Parameters:
- {buf} (integer|string)

Return:
(string)

synID({lnum}, {col}, {trans}) synID()

Returns the syntax ID at position {lnum} and {col} in the
current window. The syntax ID can be used with synIDattr()
and synIDtrans() to obtain syntax information about text.

{col} is 1 for the leftmost column, {lnum} is 1 for the first
line. 'synmaxcol' applies, in a longer line zero is returned.
Note that when the position is after the last character,
that's where the cursor can be in Insert mode, synID() returns
zero. {lnum} is used like with getline().

When {trans} is TRUE, transparent items are reduced to the
item that they reveal. This is useful when wanting to know
the effective color. When {trans} is FALSE, the transparent
item is returned. This is useful when wanting to know which
syntax item is effective (e.g. inside parens).
Warning: This function can be very slow. Best speed is
obtained by going through the file in forward direction.

Returns zero on error.

Example (echoes the name of the syntax item under the cursor):

```
echo synIDattr(synID(line("."), col("."), 1), "name")
```

Parameters:
- {lnum} (integer|string)

- {col} (integer)

- {trans} (0|1)

Return:
(integer)

synIDattr({synID}, {what} [, {mode}]) synIDattr()

Returns the {what} attribute of syntax ID {synID} as a
String. This can be used to obtain information about a
syntax item.
{mode} can be "gui" or "cterm", to get the attributes
for that mode. When {mode} is omitted, or an invalid value is
used, the attributes for the currently active highlighting are
used (GUI or cterm).
Use synIDtrans() to follow linked highlight groups.

{what}	result
"name"	the name of the syntax item
"fg"	foreground color (GUI: color name used to set the color, cterm: color number as a string, term: empty string)
"bg"	background color (as with "fg")
"font"	font name (only available in the GUI) highlight-font
"sp"	special color (as with "fg") guisp
"fg#"	like "fg", but for the GUI and the GUI is running the name in "#RRGGBB" form
"bg#"	like "fg#" for "bg"
"sp#"	like "fg#" for "sp"
"bold"	"1" if bold
"italic"	"1" if italic
"reverse"	"1" if reverse
"inverse"	"1" if inverse (= reverse)
"standout"	"1" if standout
"underline"	"1" if underlined
"undercurl"	"1" if undercurled
"underdouble"	"1" if double underlined
"underdotted"	"1" if dotted underlined
"underdashed"	"1" if dashed underlined
"strikethrough"	"1" if struckthrough
"altfont"	"1" if alternative font
"nocombine"	"1" if nocombine
"dim"	"1" if half-bright/dimmed
"blink"	"1" if blinking
"conceal"	"1" if concealed
"overline"	"1" if overlined

Returns an empty string on error.

Example (echoes the color of the syntax item under the
cursor):

```
echo synIDattr(synIDtrans(synID(line("."), col("."), 1)), "fg")
```

Can also be used as a method:

```
echo synID(line("."), col("."), 1)->synIDtrans()->synIDattr("fg")
```

Parameters:
- {synID} (integer)

- {what} (string)

- {mode} (string?)

Return:
(string)

synIDtrans({synID}) synIDtrans()

Returns the translated syntax ID of {synID}, following
highlight links. This is the syntax group ID of what is
being used to highlight the character. Highlight links given with
":highlight link" are followed.

Returns zero on error.

Parameters:
- {synID} (integer)

Return:
(integer)

synconcealed({lnum}, {col}) synconcealed()

Returns conceal information for the character at position {lnum} and {col} as a List with three items:

1. The first item in the list is 0 if the character is not part of a concealable region, 1 if it is. {lnum} is used like with getline().
2. The second item in the list is a string. If the first item is 1, the second item contains the text which will be displayed in place of the concealed text, depending on the current setting of 'conceallevel' and 'listchars'.
3. The third and final item in the list is a number representing the specific syntax region matched in the line. When the character is not concealed the value is zero. This allows detection of the beginning of a new concealable region if there are two consecutive regions with the same replacement character. For an example, if the text is "123456" and both "23" and "45" are concealed and replaced by the character "X", then:

call	returns ~
synconcealed(lnum, 1)	[0, '', 0]
synconcealed(lnum, 2)	[1, 'X', 1]
synconcealed(lnum, 3)	[1, 'X', 1]
synconcealed(lnum, 4)	[1, 'X', 2]
synconcealed(lnum, 5)	[1, 'X', 2]
synconcealed(lnum, 6)	[0, '', 0]

Note: Doesn't consider matchadd() highlighting items, since syntax and matching highlighting are two different mechanisms syntax-vs-match.

Parameters:

- {lnum} (integer|string)
- {col} (integer)

Return:

([integer, string, integer])

synstack({lnum}, {col}) synstack()

Return a List, which is the stack of syntax items at the position {lnum} and {col} in the current window. {lnum} is used like with getline(). Each item in the List is an ID like what synID() returns.
The first item in the List is the outer region, following are items contained in that one. The last one is what synID() returns, unless not the whole item is highlighted or it is a transparent item.
This function is useful for debugging a syntax file.
Example that shows the syntax stack under the cursor:

```
for id in synstack(line("."), col("."))
   echo synIDattr(id, "name")
endfor
```

When the position specified with {lnum} and {col} is invalid an empty list is returned. The position just after the last character in a line and the first column in an empty line are valid positions.

Parameters:

- {lnum} (integer|string)
- {col} (integer)

Return:

(integer[])

system({cmd} [, {input}]) system() E677

Lua: Prefer vim.system(...):wait().stdout for string output; string commands use the shell in Vimscript, not by default in Lua.

Gets the output of {cmd} as a string (systemlist() returns a List) and sets v:shell_error to the error code.
{cmd} is treated as in jobstart():
If {cmd} is a List it runs directly (no 'shell').
If {cmd} is a String it runs in the 'shell', like this:

```
call jobstart(split(&shell) + split(&shellcmdflag) + ['{cmd}'])
```

Not to be used for interactive commands.

Result is a String, filtered to avoid platform-specific quirks:

- <CR><NL> is replaced with <NL>

- NUL characters are replaced with SOH (0x01)

Example:

```
echo system(['ls', expand('%:h')])
```

If {input} is a string it is written to a pipe and passed as stdin to the command. The string is written as-is, line separators are not changed.
If {input} is a List it is written to the pipe as writefile() does with {binary} set to "b" (i.e. with a newline between each list item, and newlines inside list items converted to NULs).
When {input} is given and is a valid buffer id, the content of the buffer is written to the file line by line, each line terminated by NL (and NUL where the text has NL).

E5677

Note: system() cannot write to or read from backgrounded ("&") shell commands, e.g.:

```
echo system("cat - &", "foo")
```

which is equivalent to:

```
$ echo foo | bash -c 'cat - &'
```

The pipes are disconnected (unless overridden by shell redirection syntax) before input can reach it. Use jobstart() instead.

Note: Use shellescape() or ::S with expand() or fnamemodify() to escape special characters in a command argument. 'shellquote' and 'shellxquote' must be properly configured. Example:

```
echo system('ls '..shellescape(expand('%:h')))
echo system('ls '..expand('%:h:S'))
```

Unlike ":!cmd" there is no automatic check for changed files. Use :checktime to force a check.

Parameters:
- {cmd} (string|string[])

- {input} (string|string[]|integer?)

Return:
(string)

systemlist({cmd} [, {input} [, {keepempty}]]) systemlist()

Lua: Prefer vim.system(...):wait().stdout split into lines; string commands use the shell in Vimscript, not by default in Lua.

Same as system(), but returns a List with lines (parts of output separated by NL) with NULs transformed into NLs. Output is the same as readfile() will output with {binary} argument set to "b", except that a final newline is not preserved, unless {keepempty} is non-zero.
Note that on MS-Windows you may get trailing CR characters.

To see the difference between "echo hello" and "echo -n hello" use system() and split():

```
echo split(system('echo hello'), '\n', 1)
```

Returns an empty string on error.

Parameters:
- {cmd} (string|string[])

- {input} (string|string[]|integer?)

- {keepempty} (integer?)

Return:
(string[])

tabpagebuflist([{arg}]) tabpagebuflist()

Lua: Prefer nvim_tabpage_list_wins() with

nvim_win_get_buf(); invalid tabpages error instead of
returning 0.

Returns a List of buffer numbers, one for each window in
the specified tabpage.
{arg} specifies the number of the tabpage to be used. When
omitted the current tabpage is used.
When {arg} is invalid the number zero is returned.
To get a list of all buffers in all tabs use this:

```
let buflist = []
for i in range(tabpagenr('$'))
   call extend(buflist, tabpagebuflist(i + 1))
endfor
```

Note that a buffer may appear in more than one window.

Parameters:
- {arg} (integer?)

Return:
(any)

tabpagenr([{arg}]) tabpagenr()

Lua: Prefer nvim_tabpage_get_number() with
nvim_get_current_tabpage() for the current tab, or
#vim.api.nvim_list_tabpages() for "$"; "#" differs.

Returns the current tabpage-number.

The optional argument {arg} supports the following values:
  $ the last tabpage-number (the tabpage count).
  #   the last-accessed tabpage-number (where g<Tab>
        goes to), or 0 if there is no previous tabpage.
The number can be used with the :tab command.

Returns zero on error.

Parameters:
- {arg} ('$'|'#'?)

Return:
(integer)

tabpagewinnr({tabarg} [, {arg}]) tabpagewinnr()

Lua: Prefer nvim_win_get_number() with
nvim_tabpage_get_win() for the current window, or the length
of nvim_tabpage_list_wins() for "$"; "#" and invalid-tab
behavior differ.

Like winnr() but for tabpage {tabarg}.
{tabarg} specifies the tabpage-number to be used.
{arg} is used like with winnr():
- When omitted the current window number is returned. This is
  the window which will be used when going to this tabpage.
- When "$" the number of windows is returned.
- When "#" the previous window nr is returned.

Useful examples:

```
tabpagewinnr(1)     " current window of tabpage 1
tabpagewinnr(4, '$')    " number of windows in tabpage 4
```

When {tabarg} is invalid zero is returned.

Parameters:
- {tabarg} (integer)
- {arg} ('$'|'#'?)

Return:
(integer)

tagfiles() tagfiles()

Returns a List with the file names used to search for tags
for the current buffer. This is the 'tags' option expanded.

Return:
(string[])

taglist({expr} [, {filename}]) taglist()

Returns a List of tags matching the regular expression {expr}.

If {filename} is passed it is used to prioritize the results in the same way that :tselect does. See tag-priority. {filename} should be the full path of the file.

Each list item is a dictionary with at least the following entries:

name	Name of the tag.
filename	Name of the file where the tag is defined. It is either relative to the current directory or a full path.
cmd	Ex command used to locate the tag in the file.
kind	Type of the tag. The value for this entry depends on the language specific kind values. Only available when using a tags file generated by Universal/Exuberant ctags or hdrtag.
static	A file specific tag. Refer to static-tag for more information.

More entries may be present, depending on the content of the tags file: access, implementation, inherits and signature. Refer to the ctags documentation for information about these fields. For C code the fields "struct", "class" and "enum" may appear, they give the name of the entity the tag is contained in.

The ex-command "cmd" can be either an ex search pattern, a line number or a line number followed by a byte number.

If there are no matching tags, then an empty list is returned.

To get an exact tag match, the anchors '^' and '$' should be used in {expr}. This also make the function work faster. Refer to tag-regexp for more information about the tag search regular expression pattern.

Refer to 'tags' for information about how the tags file is located by Vim. Refer to tags-file-format for the format of the tags file generated by the different ctags tools.

Parameters:

- {expr} (any)
- {filename} (string?)

Return:

(any)

tan({expr}) tan()

Lua: Prefer math.tan().

Return the tangent of {expr}, measured in radians, as a Float in the range [-inf, inf].
{expr} must evaluate to a Float or a Number.
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo tan(10)
```

```
0.648361
```

```
echo tan(-4.01)
```

```
-1.181502
```

Parameters:

- {expr} (number)

Return:

(number)

tanh({expr}) tanh()

Lua: Prefer math.tanh().

Return the hyperbolic tangent of {expr} as a Float in the range [-1, 1].
{expr} must evaluate to a Float or a Number.
Returns 0.0 if {expr} is not a Float or a Number.
Examples:

```
echo tanh(0.5)
```

```
0.462117
```

```
echo tanh(-1)
```

```
-0.761594
```

Parameters:

- {expr} (number)

Return:
(number)

tempname() tempname()

Lua: Prefer os.tmpname() for a temporary name; Nvim's tempdir and nonexistence are not guaranteed.

Generates a (non-existent) filename located in the Nvim root tempdir. Scripts can use the filename as a temporary file. Example:

```
let tmpfile = tempname()
exe "redir > " .. tmpfile
```

Return:
(string)

test_garbagecollect_now() test_garbagecollect_now()

Like garbagecollect(), but executed right away. This must only be called directly to avoid any structure to exist internally, and v:testing must have been set before calling any function. E1142

Return:
(any)

timer_info([{id}]) timer_info()

Return a list with information about timers.
When {id} is given only information about this timer is returned. When timer {id} does not exist an empty list is returned.
When {id} is omitted information about all timers is returned.

For each timer the information is stored in a Dictionary with these items:

"id"	the timer ID
"time"	time the timer was started with
"repeat"	number of times the timer will still fire; -1 means forever
"callback"	the callback

Parameters:
- {id} (integer?)

Return:
(any)

timer_pause({timer}, {paused}) timer_pause()

Pause or unpause a timer. A paused timer does not invoke its callback when its time expires. Unpausing a timer may cause the callback to be invoked almost immediately if enough time has passed.

Pausing a timer is useful to avoid the callback to be called for a short time.

If {paused} evaluates to a non-zero Number or a non-empty String, then the timer is paused, otherwise it is unpaused. See non-zero-arg.

Parameters:
- {timer} (integer)
- {paused} (boolean)

Return:
(any)

timer_start({time}, {callback} [, {options}]) timer_start() timer

Lua: Prefer vim.defer_fn() for one-shot timers, or uv.new_timer() for repeating timers; callback and handle semantics differ.

Create a timer and return the timer ID.

{time} is the waiting time in milliseconds. This is the minimum time before invoking the callback. When the system is busy or Vim is not waiting for input the time will be longer. Zero can be used to execute the callback when Vim is back in the main loop.

{callback} is the function to call. It can be the name of a function or a Funcref. It is called with one argument, which is the timer ID. The callback is only invoked when Vim is waiting for input.

{options} is a dictionary. Supported entries:

"repeat"	Number of times to repeat the callback. -1 means forever. Default is 1.

If the timer causes an error three times in a row the repeat is cancelled.

Returns -1 on error.

Example:

```
func MyHandler(timer)
  echo 'Handler called'
endfunc
let timer = timer_start(500, 'MyHandler',
        \ {'repeat': 3})
```

This invokes MyHandler() three times at 500 msec intervals.

Parameters:
- {time} (number)
- {callback} (string|function)
- {options} (table?)

Return:
(any)

timer_stop({timer}) timer_stop()

Lua: Prefer uv.timer_stop() and uv.close() on a stored uv.new_timer() handle; timer IDs differ.

Stop a timer. The timer callback will no longer be invoked. {timer} is an ID returned by timer_start(), thus it must be a Number. If {timer} does not exist there is no error.

Parameters:
- {timer} (integer)

Return:
(any)

timer_stopall() timer_stopall()

Stop all timers. The timer callbacks will no longer be invoked. Useful if some timers is misbehaving. If there are no timers there is no error.

Return:
(any)

tolower({expr}) tolower()

Lua: Prefer string.lower().

Converts a String to lowercase (like applying gu). Returns empty string on error.

Attributes:
api-fast

Parameters:
- {expr} (string)

Return:
(string)

toupper({expr}) toupper()

Lua: Prefer string.upper().

Converts a String to uppercase (like applying gU). Returns empty string on error.

Attributes:
api-fast

Parameters:
- {expr} (string)

Return:
(string)

tr({src}, {fromstr}, {tostr}) tr()

Translates characters in {src}, replacing each character that appears in {fromstr} with the corresponding character in {tostr}. Exactly like the unix "tr" command.
This code also deals with multibyte characters properly.

Returns an empty string on error.

Examples:

```
echo tr("hello there", "ht", "HT")
```

returns "Hello THere"

```
echo tr("<blob>", "<>", "{}")
```

returns "{blob}"

Attributes:
api-fast

Parameters:
- {src} (string)
- {fromstr} (string)
- {tostr} (string)

Return:
(string)

trim({text} [, {mask} [, {dir}]]) trim()

Lua: Prefer vim.trim().

Return {text} as a String where any character in {mask} is removed from the beginning and/or end of {text}.

If {mask} is not given, or is an empty string, {mask} is all characters up to 0x20, which includes Tab, space, NL and CR, plus the non-breaking space character 0xa0.

The optional {dir} argument specifies where to remove the characters:
0 remove from the beginning and end of {text}
1 remove only at the beginning of {text}
2 remove only at the end of {text}
When omitted both ends are trimmed.

This function deals with multibyte characters properly. Returns an empty string on error.

Examples:

```
echo trim("   some text ")
```

returns "some text"

```
echo trim("  \r\t\t\r RESERVE \t\n\x0B\xA0") .. "_TAIL"
```

returns "RESERVE_TAIL"

```
echo trim("rm<Xrm<>X>rrm", "rm<>")
```

returns "Xrm<>X" (characters in the middle are not removed)

```
echo trim("  vim  ", " ", 2)
```

returns " vim"

Attributes:
api-fast

Parameters:
- {text} (string)
- {mask} (string?)
- {dir} (0|1|2?)

Return:
(string)

trunc({expr}) trunc()

Lua: Prefer math.modf() (first return value).

Return the largest integral value with magnitude less than or equal to {expr} as a Float (truncate towards zero). {expr} must evaluate to a Float or a Number. Returns 0.0 if {expr} is not a Float or a Number. Examples:

```
echo trunc(1.456)
```

1.0

```
echo trunc(-5.456)
```

-5.0

```
echo trunc(4.0)

                    4.0
```

Parameters:
- {expr} (number)

Return:
(integer)

type({expr}) type()

Lua: Prefer lua-type() for most Lua type checks; Vim type IDs differ.

Returns the type of {expr} as a Number.
Instead of using the number directly, it is better to use the v:t_ variable that has the value:

Number:	0	v:t_number
String:	1	v:t_string
Funcref:	2	v:t_func
List:	3	v:t_list
Dictionary:	4	v:t_dict
Float:	5	v:t_float
Boolean:	6	v:t_bool (v:false and v:true)
Null:	7	(v:null)
Blob:	10	v:t_blob

For backward compatibility, this method can be used:

```
if type(myvar) == type(0) | endif
if type(myvar) == type("") | endif
if type(myvar) == type(function("tr")) | endif
if type(myvar) == type([]) | endif
if type(myvar) == type({}) | endif
if type(myvar) == type(0.0) | endif
if type(myvar) == type(v:true) | endif
```

In place of checking for v:null type it is better to check for v:null directly as it is the only value of this type:

```
if myvar is v:null | endif
```

To check if the v:t_ variables exist use this:

```
if exists('v:t_number') | endif
```

Attributes:
api-fast

Parameters:
- {expr} (any)

Return:
(integer)

undofile({name}) undofile()

Return the name of the undo file that would be used for a file with name {name} when writing. This uses the 'undodir' option, finding directories that exist. It does not check if the undo file exists.
{name} is always expanded to the full path, since that is what is used internally.
If {name} is empty undofile() returns an empty string, since a buffer without a file name will not write an undo file.
Useful in combination with :wundo and :rundo.

Parameters:
- {name} (string)

Return:
(string)

undotree([{buf}]) undotree()

Return the current state of the undo tree for the current buffer, or for a specific buffer if {buf} is given. The result is a dictionary with the following items:

"seq_last"	The highest undo sequence number used.
"seq_cur"	The sequence number of the current position in the undo tree. This differs from "seq_last" when some changes were undone.
"time_cur"	Time last used for :earlier and related commands. Use strftime() to convert to something readable.
"save_last"	Number of the last file write. Zero when no write yet.
"save_cur"	Number of the current position in the undo tree.

"synced" Non-zero when the last undo block was synced. This happens when waiting from input from the user. See undo-blocks.
"entries" A list of dictionaries with information about undo blocks.

The first item in the "entries" list is the oldest undo item. Each List item is a Dictionary with these items:
"seq" Undo sequence number. Same as what appears in :undolist.
"time" Timestamp when the change happened. Use strftime() to convert to something readable.
"newhead" Only appears in the item that is the last one that was added. This marks the last change and where further changes will be added.
"curhead" Only appears in the item that is the last one that was undone. This marks the current position in the undo tree, the block that will be used by a redo command. When nothing was undone after the last change this item will not appear anywhere.
"save" Only appears on the last block before a file write. The number is the write count. The first write has number 1, the last one the "save_last" mentioned above.
"alt" Alternate entry. This is again a List of undo blocks. Each item may again have an "alt" item.

Parameters:
- {buf} (integer|string?)

Return:
(vim.fn.undotree.ret)

uniq({list} [, {func} [, {dict}]]) uniq() E882

Lua: Prefer vim.list.unique().

Remove second and succeeding copies of repeated adjacent {list} items in-place. Returns {list}. If you want a list to remain unmodified make a copy first:

```
let newlist = uniq(copy(mylist))
```

The default compare function uses the string representation of each item. For the use of {func} and {dict} see sort(). For deduplicating text in the current buffer see :uniq.

Returns zero if {list} is not a List.

Parameters:
- {list} (any)
- {func} (any?)
- {dict} (any?)

Return:
(any[]|0)

utf16idx({string}, {idx} [, {countcc} [, {charidx}]]) utf16idx()

Lua: Prefer vim.str_utfindex() with "utf-16" and strict_indexing = false; countcc and charidx semantics differ.

Same as charidx() but returns the UTF-16 code unit index of the byte at {idx} in {string} (after converting it to UTF-16).

When {charidx} is present and TRUE, {idx} is used as the character index in the String {string} instead of as the byte index.
An {idx} in the middle of a UTF-8 sequence is rounded downwards to the beginning of that sequence.

Returns -1 if the arguments are invalid or if there are less than {idx} bytes in {string}. If there are exactly {idx} bytes, the length of the string in UTF-16 code units is returned.

See byteidx() and byteidxcomp() for getting the byte index from the UTF-16 index and charidx() for getting the character index from the UTF-16 index.
Refer to string-offset-encoding for more information.
Examples:

```
echo utf16idx('a😊😊', 3)       " returns 2
echo utf16idx('a😊😊', 7)       " returns 4
echo utf16idx('a😊😊', 1, 0, 1) " returns 2
echo utf16idx('a😊😊', 2, 0, 1) " returns 4
echo utf16idx('aą́c', 6)         " returns 2
echo utf16idx('aą́c', 6, 1)      " returns 4
echo utf16idx('a😊😊', 9)       " returns -1
```

Attributes:
api-fast

Parameters:
- {string} (string)
- {idx} (integer)
- {countcc} (boolean?)
- {charidx} (boolean?)

Return:
(integer)

values({dict}) values()

Lua: Prefer vim.tbl_values().

Return a List with all the values of {dict}. The List is in arbitrary order. Also see items() and keys(). Returns zero if {dict} is not a Dict.

Parameters:
- {dict} (any)

Return:
(any)

virtcol({expr} [, {list} [, {winid}]]) virtcol()

Returns the virtual (screen) column of the file position given with {expr}. That is, the total number of screen cells occupied by the part of the line until the end of the character at that position. When there is a <Tab> at the position, the returned Number will be the column at the end of the <Tab>. For example, for a <Tab> in column 1, with 'ts' set to 8, it returns 8. conceal is ignored. For the byte position use col().

For the use of {expr} see getpos() and col(). When {expr} is "$", it means the end of the cursor line, so the result is the number of cells in the cursor line plus one.

When 'virtualedit' is used {expr} can be [lnum, col, off], where "off" is the offset in screen columns from the start of the character. E.g., a position within a <Tab> or after the last character. When "off" is omitted zero is used. When Virtual editing is active in the current mode, a position beyond the end of the line can be returned. Also see 'virtualedit'

If {list} is present and non-zero then virtcol() returns a List with the first and last screen position occupied by the character.

With the optional {winid} argument the values are obtained for that window instead of the current window.

Note that only marks in the current file can be used. Examples:

```
" With text "foo^Lbar" and cursor on the "^L":

echo virtcol(".")       " returns 5
echo virtcol(".", 1)    " returns [4, 5]
echo virtcol("$")       " returns 9

" With text "     there", with 't at 'h':

echo virtcol("'t")      " returns 6
```

The first column is 1. 0 or [0, 0] is returned for an error.

A more advanced example that echoes the maximum length of all lines:

```
echo max(map(range(1, line('$')), "virtcol([v:val, '$'])"))
```

Parameters:
- {expr} (string|any[])

- {list} (boolean?)

- {winid} (integer?)

Return:
(integer|[integer, integer])

virtcol2col({winid}, {lnum}, {col}) virtcol2col()

Converts a virtual column to a byte index. Returns the byte index of the character in window {winid} at buffer line {lnum} and virtual column {col}.

If buffer line {lnum} is an empty line, 0 is returned.

If {col} is greater than the last virtual column in line {lnum}, then the byte index of the character at the last virtual column is returned.

For a multi-byte character, the column number of the first byte in the character is returned.

{winid} is a window-number or window-ID. If zero, the current window is used.

Returns -1 if the window {winid} doesn't exist or the buffer line {lnum} or virtual column {col} is invalid.

See also screenpos(), virtcol() and col().

Parameters:
- {winid} (integer)

- {lnum} (integer)

- {col} (integer)

Return:
(integer)

visualmode([{expr}]) visualmode()

Returns a String describing the last Visual mode used in the current buffer. Initially it returns an empty string, but once Visual mode has been used, it returns "v", "V", or "<CTRL-V>" (a single CTRL-V character) for character-wise, line-wise, or block-wise Visual mode respectively.
Example:

```
exe "normal " .. visualmode()
```

This enters the same Visual mode as before. It is also useful in scripts if you wish to act differently depending on the Visual mode that was used.
If Visual mode is active, use mode() to get the Visual mode (e.g., in a :vmap).
If {expr} is supplied and it evaluates to a non-zero Number or a non-empty String, then the Visual mode will be cleared and the old value is returned. See non-zero-arg.

Parameters:
- {expr} (boolean?)

Return:
(string)

wait({timeout}, {condition} [, {interval}]) wait()

Lua: Prefer vim.wait().

Waits until {condition} evaluates to TRUE, where {condition} is a Funcref or string containing an expression.

{timeout} is the maximum waiting time in milliseconds, -1 means forever.

Condition is evaluated on user events, internal events, and every {interval} milliseconds (default: 200).

Returns a status integer:
	0 if the condition was satisfied before timeout
	-1 if the timeout was exceeded
	-2 if the function was interrupted (by CTRL-C)
	-3 if an error occurred

Parameters:
- {timeout} (integer)

- {condition} (any)

- {interval} (number?)

Return:
(any)

wildmenumode() wildmenumode()

Returns TRUE when the wildmenu is active and FALSE
otherwise. See 'wildmenu' and 'wildmode'.
This can be used in mappings to handle the 'wildcharm' option
gracefully. (Makes only sense with mapmode-c mappings).

For example to make <c-j> work like <down> in wildmode, use:

```
cnoremap <expr> <C-j> wildmenumode() ? "\<Down>\<Tab>" : "\<c-j>"
```

(**Note:** this needs the 'wildcharm' option set appropriately).

Return:
(any)

wildtrigger() wildtrigger()

Start wildcard expansion in the command-line, using the
behavior defined by the 'wildmode' and 'wildoptions' settings.

This function also enables completion in search patterns such
as /, ?, :s, :g, :v and :vimgrep.

Unlike pressing 'wildchar' manually, this function does not
produce a beep when no matches are found and generally
operates more quietly. This makes it suitable for triggering
completion automatically.

Note: After navigating command-line history, the first call to
wildtrigger() is a no-op; a second call is needed to start
expansion. This is to support history navigation in
command-line autocompletion.

See cmdline-autocompletion.

Return value is always 0.

Return:
(number)

win_execute({id}, {command} [, {silent}]) win_execute()

Lua: Prefer nvim_win_call().

Like execute() but in the context of window {id}.
The window will temporarily be made the current window,
without triggering autocommands or changing directory. When
executing {command} autocommands will be triggered, this may
have unexpected side effects. Use :noautocmd if needed.
Example:

```
call win_execute(winid, 'syntax enable')
```

Doing the same with setwinvar() would not trigger
autocommands and not actually show syntax highlighting.

When window {id} does not exist then no error is given and
an empty string is returned.

Parameters:
- {id} (integer)

- {command} (string)

- {silent} (boolean?)

Return:
(any)

win_findbuf({bufnr}) win_findbuf()

Lua: Prefer nvim_list_wins() with nvim_win_get_buf().

Returns a List with window-IDs for windows that contain
buffer {bufnr}. When there is none the list is empty.

Parameters:
- {bufnr} (integer)

Return:
(integer[])

win_getid([{win} [, {tab}]]) win_getid()

Lua: Prefer nvim_get_current_win() when omitted, or

nvim_tabpage_list_wins() with nvim_win_get_number() after
resolving {tab}; invalid windows return 0 in Vimscript.

Get the window-ID for the specified window.
When {win} is missing use the current window.
With {win} this is the window-number.
Without {tab} use the current tab, otherwise the
tabpage-number given by {tab}.
Return zero if the window cannot be found.

Parameters:
- {win} (integer?)

- {tab} (integer?)

Return:
(integer)

win_gettype([{nr}]) win_gettype()

Gets the type of the given window, or current window if {nr}
is omitted:
- (empty) Normal window

- "autocmd" Internal "context-switch" window.

- "command" Command-line window cmdwin

- "loclist" location-list-window

- "popup" Floating window api-floatwin

- "preview" Preview window preview-window

- "quickfix" quickfix-window

- "unknown" Window {nr} not found

See also the 'buftype' option.

Parameters:
- {nr} (integer?)

Return:
('autocmd'|'command'|''|'loclist'|'popup'|'preview'|'quickfix'|'unknown')

win_gotoid({expr}) win_gotoid()

Lua: Prefer nvim_set_current_win(); invalid IDs error
instead of returning FALSE.

Go to window-ID {expr}. May change the current tabpage.
Return TRUE if successful, FALSE if the window cannot be
found.

Parameters:
- {expr} (integer)

Return:
(0|1)

win_id2tabwin({expr}) win_id2tabwin()

Lua: Prefer nvim_win_get_tabpage() and
nvim_win_get_number() for valid windows; invalid IDs error
instead of returning [0, 0].

Return [tabnr, winnr] for window with window-ID {expr}, where
tabnr is the tabpage-number and winnr is the window-number.
Return [0, 0] if the window cannot be found.

Parameters:
- {expr} (integer)

Return:
(any)

win_id2win({expr}) win_id2win()

Lua: Prefer nvim_win_get_number() for valid windows; windows
outside the current tabpage and invalid IDs differ.

Return the window-number of for the given window-ID {expr}.
Return 0 if the window cannot be found in the current tabpage.

Parameters:
- {expr} (integer)

Return:
(integer)

win_move_separator({nr}, {offset}) win_move_separator()

Move window {nr}'s vertical separator (i.e., the right border) by {offset} columns, as if being dragged by the mouse. {nr} is a window-number or window-ID. A positive {offset} moves right and a negative {offset} moves left. Moving a window's vertical separator will change the width of the window and the width of other windows adjacent to the vertical separator. The magnitude of movement may be smaller than specified (e.g., as a consequence of maintaining 'winminwidth'). Returns TRUE if the window can be found and FALSE otherwise.
This will fail for the rightmost window and a full-width window, since it has no separator on the right.
Only works for the current tabpage. E1308

Parameters:
- {nr} (integer)
- {offset} (integer)

Return:
(any)

win_move_statusline({nr}, {offset}) win_move_statusline()

Move window {nr}'s status line (i.e., the bottom border) by {offset} rows, as if being dragged by the mouse. {nr} is a window-number or window-ID. A positive {offset} moves down and a negative {offset} moves up. Moving a window's status line will change the height of the window and the height of other windows adjacent to the status line. The magnitude of movement may be smaller than specified (e.g., as a consequence of maintaining 'winminheight'). Returns TRUE if the window can be found and FALSE otherwise.
Only works for the current tabpage.

Parameters:
- {nr} (integer)
- {offset} (integer)

Return:
(any)

win_screenpos({nr}) win_screenpos()

Lua: Prefer nvim_win_get_position() and add 1 to row and col; invalid-window semantics differ.

Return the screen position of window {nr} as a list with two numbers: [row, col]. The first window always has position [1, 1], unless there is a tabline, then it is [2, 1].
{nr} is a window-number or window-ID. Use zero for the current window.
Returns [0, 0] if the window cannot be found.

Parameters:
- {nr} (integer)

Return:
(any)

win_splitmove({nr}, {target} [, {options}]) win_splitmove()

Temporarily switch to window {target}, then move window {nr} to a new split adjacent to {target}.
Unlike commands such as :split, no new windows are created (the window-ID of window {nr} is unchanged after the move).

Both {nr} and {target} are a window-number or window-ID.
Both must be in the current tabpage.

Returns zero for success, non-zero for failure.

{options} is a Dictionary with the following optional entries:
- "vertical" When TRUE, the split is created vertically, like with :vsplit.
- "rightbelow" When TRUE, the split is made below or to the right (if vertical). When FALSE, it is done

above or to the left (if vertical). When not present, the values of 'splitbelow' and 'splitright' are used.

Parameters:
- {nr} (integer)
- {target} (integer)
- {options} (table?)

Return:
(any)

winbufnr({nr}) winbufnr()

Lua: Prefer nvim_win_get_buf().

Returns the buffer number associated with window {nr} (window-number or window-ID).
When {nr} is zero, the number of the buffer in the current window is returned.
When window {nr} doesn't exist, -1 is returned.
Example:

```
echo "The file in the current window is " .. bufname(winbufnr(0))
```

Parameters:
- {nr} (integer)

Return:
(integer)

wincol() wincol()

Returns the virtual column of the cursor in the window. This is counting screen cells from the left side of the window. The leftmost column is one.

Return:
(integer)

windowsversion() windowsversion()

Returns the Windows OS version as a String. E.g, Windows 10 is "10.0", Windows 8 is "6.2", Windows XP is "5.1". For non-MS-Windows systems the result is an empty string.

Lua: see uv.os_uname().

Attributes:
api-fast

Return:
(string)

winheight({nr}) winheight()

Gets the height of window-ID {nr} (zero for "current window"), excluding any 'winbar' and 'statusline'. Returns -1 if window {nr} doesn't exist. An existing window always has a height of zero or more.

Examples:

```
echo "Current window has " .. winheight(0) .. " lines."
```

Parameters:
- {nr} (integer)

Return:
(integer)

winlayout([{tabnr}]) winlayout()

Returns the layout of windows in a tabpage as a nested List.

Without {tabnr} use the current tabpage, otherwise the tabpage with number {tabnr}. If the tabpage {tabnr} is not found, returns an empty list.

For a leaf window, it returns:

```
["leaf", {winid}]
```

For horizontally split windows, which form a column, it returns:

```
["col", [{nested list of windows}]]
```

For vertically split windows, which form a row, it returns:

```
["row", [{nested list of windows}]]
```

Example:

```
" Only one window in the tabpage
echo winlayout()
```

```
['leaf', 1000]
```

```
" Two horizontally split windows
echo winlayout()
```

```
['col', [['leaf', 1000], ['leaf', 1001]]]
```

```
" The second tabpage, with three horizontally split
" windows, with two vertically split windows in the
" middle window
echo winlayout(2)
```

```
['col', [['leaf', 1002], ['row', [['leaf', 1003],
                    ['leaf', 1001]]], ['leaf', 1000]]]
```

Parameters:
- {tabnr} (integer?)

Return:
(vim.fn.winlayout.ret)

winline() winline()

Returns the screen line of the cursor in the window. This is counting screen lines from the top of the window. The first line is one.
If the cursor was moved the view on the file will be updated first, this may cause a scroll.

Return:
(integer)

winnr([{arg}]) winnr()

Lua: Prefer nvim_win_get_number() with nvim_get_current_win() for the current window, or nvim_tabpage_list_wins() with nvim_get_current_tabpage() for "$"; "#", motion args, and hidden windows differ.

Returns the number of the current window. The top window has number 1. Returns zero for a hidden or non focusable window, unless it is the current window.

The optional argument {arg} supports the following values:

$	the number of the last window (the window count).
#	the number of the last accessed window (where CTRL-W_p goes to). If there is no previous window or it is in another tabpage, 0 is returned. May refer to the current window in some cases (e.g. when evaluating 'statusline' expressions).
{N}j	the number of the Nth window below the current window (where CTRL-W_j goes to).
{N}k	the number of the Nth window above the current window (where CTRL-W_k goes to).
{N}h	the number of the Nth window left of the current window (where CTRL-W_h goes to).
{N}l	the number of the Nth window right of the current window (where CTRL-W_l goes to).

The number can be used with CTRL-W_w and ":wincmd w" :wincmd.
When {arg} is invalid an error is given and zero is returned.
Also see tabpagewinnr() and win_getid().
Examples:

```
let window_count = winnr('$')
let prev_window = winnr('#')
let wnum = winnr('3k')
```

Parameters:
- {arg} (string|integer?)

Return:
(integer)

winrestcmd() winrestcmd()

Returns a sequence of :resize commands that should restore the current window sizes. Only works properly when no windows are opened or closed and the current window and tabpage is

unchanged.
Example:

```
let cmd = winrestcmd()
call MessWithWindowSizes()
exe cmd
```

Return:
(string)

winrestview({dict}) winrestview()

Uses the Dictionary returned by winsaveview() to restore the view of the current window.
Note: The {dict} does not have to contain all values, that are returned by winsaveview(). If values are missing, those settings won't be restored. So you can use:

```
call winrestview({'curswant': 4})
```

This will only set the curswant value (the column the cursor wants to move on vertical movements) of the cursor to column 5 (yes, that is 5), while all other settings will remain the same. This is useful, if you set the cursor position manually.

If you have changed the values the result is unpredictable. If the window size changed the result won't be the same.

Parameters:
- {dict} (vim.fn.winrestview.dict)

Return:
(any)

winsaveview() winsaveview()

Returns a Dictionary that contains information to restore the view of the current window. Use winrestview() to restore the view.
This is useful if you have a mapping that jumps around in the buffer and you want to go back to the original view.
This does not save fold information. Use the 'foldenable' option to temporarily switch off folding, so that folds are not opened when moving around. This may have side effects.
The return value includes:

lnum	cursor line number
col	cursor column (**Note:** the first column zero, as opposed to what getcurpos() returns)
coladd	cursor column offset for 'virtualedit'
curswant	column for vertical movement (**Note:** the first column is zero, as opposed to what getcurpos() returns). After $ command it will be a very large number equal to v:maxcol.
topline	first line in the window
topfill	filler lines, only in diff mode
leftcol	first column displayed; only used when 'wrap' is off
skipcol	columns skipped

Note that no option values are saved.

Return:
(vim.fn.winsaveview.ret)

winwidth({nr}) winwidth()

Gets the width of window-ID {nr} (zero for "current window"), including columns (sign-column, 'statuscolumn', etc.). Returns -1 if window {nr} doesn't exist. An existing window always has a width of zero or more.

Example:

```
echo "Current window has " .. winwidth(0) .. " columns."
if winwidth(0) <= 50
  50 wincmd |
endif
```

To get the buffer "viewport", use getwininfo():

```
:echo getwininfo(win_getid())[0].width - getwininfo(win_getid())[0].textoff
```

To get the Nvim screen size, see the 'columns' option.

Parameters:
- {nr} (integer)

Return:
(integer)

wordcount() wordcount()

Returns a dictionary of byte/chars/word statistics for the current buffer. This is the same info provided by g_CTRL-G. The return value includes:

bytes	Number of bytes in the buffer
chars	Number of chars in the buffer
words	Number of words in the buffer
cursor_bytes	Number of bytes before cursor position (not in Visual mode)
cursor_chars	Number of chars before cursor position (not in Visual mode)
cursor_words	Number of words before cursor position (not in Visual mode)
visual_bytes	Number of bytes visually selected (only in Visual mode)
visual_chars	Number of chars visually selected (only in Visual mode)
visual_words	Number of words visually selected (only in Visual mode)

Return:
(any)

writefile({data}, {fname} [, {flags}]) writefile()

Writes {data} to file {fname}.

- When {data} is a Blob its bytes are written unmodified (even if binary mode "b" is not specified).
- When {data} is a Lua string, it is treated as a blob.
- When {data} is a List, each list item is treated as a text line (terminated with a newline). Each list item must be a String or Number.
  - Any NL (newline) chars in the line are treated as a NUL character. (This is a workaround to allow Vimscript to write binary data, and is irrelevant for Lua, which should just pass a string instead.)

{flags} must be a String. These characters are recognized:

'b' Binary mode is used: There will not be a NL after the last list item. An empty item at the end does cause the last line in the file to end in a NL.

'a' Append mode is used, lines are appended to the file:

```
call writefile(["foo"], "event.log", "a")
call writefile(["bar"], "event.log", "a")
```

'D' Delete the file when the current function ends. This works like:

```
defer delete({fname})
```

Fails when not in a function. Also see :defer.

's' fsync() is called after writing the file. This flushes the file to disk, if possible. This takes more time but avoids losing the file if the system crashes.

'S' fsync() is not called, even when 'fsync' is set.

When {flags} does not contain "S" or "s" then fsync() is called if the 'fsync' option is set.

An existing file is overwritten, if possible.

When the write fails -1 is returned, otherwise 0. There is an error message if the file can't be created or when writing fails.

Also see readfile().
To copy a file byte for byte:

```
let fl = readfile("foo", "b")
call writefile(fl, "foocopy", "b")
```

Parameters:
- {data} (any)
- {fname} (string)
- {flags} (string?)

Return:
(any)

xor({expr}, {expr}) xor()

Lua: Prefer bit.bxor().

Bitwise XOR on the two arguments. The arguments are converted to a number. A List, Dict or Float argument causes an error. Also see and() and or(). Example:

```
let bits = xor(bits, 0x80)
```

Parameters:
- {expr} (integer)

- {expr1} (integer)

Return:
(integer)

2. Matching a pattern in a String string-match

This is common between several functions. A regexp pattern as explained at pattern is normally used to find a match in the buffer lines. When a pattern is used to find a match in a String, almost everything works in the same way. The difference is that a String is handled like it is one line. When it contains a "\n" character, this is not seen as a line break for the pattern. It can be matched with a "\n" in the pattern, or with ".". Example:

```
let a = "aaaa\nxxxx"
echo matchstr(a, "..\n..")
" aa
" xx
echo matchstr(a, "a.x")
" a
" x
```

Don't forget that "^" will only match at the first character of the String and "$" at the last character of the string. They don't match after or before a "\n".

Vimscript and debugging

This reference describes commands for running and debugging Vimscript.
For details on authoring Vimscript, see: expression vimscript-functions

Using Vimscript using-scripts

For writing a Vim script, see chapter 41 of the user manual usr_41.txt.

:so :source

:so[urce] {file} Runs Ex-commands or Lua code (".lua" files) from {file}. Triggers the SourcePre autocommand.

:[range]so[urce] Read Ex commands or Lua code from the [range] of lines in the current buffer. When [range] is omitted read all lines. The buffer is treated as Lua code if its 'filetype' is "lua" or its filename ends with ".lua".

When sourcing commands or Lua code from the current buffer, the same script-ID <SID> is used even if the buffer is sourced multiple times. If a buffer is sourced more than once, then the functions in the buffer are defined again.

Implementation detail: When sourcing a [range] of lines that falls inside a folded region, the range will be adjusted to the start and end of the fold, but only if a two line specifiers range was used.

:source!

:so[urce]! {file} Runs Normal-mode commands from {file}. When used after :global, :argdo, :windo, :bufdo, in a loop or when another command follows the display won't be updated while executing the commands.

:ru :runtime

:ru[ntime][!] [where] {file} ..
Sources Ex commands or Lua code (".lua" files) read from {file} (a relative path) in each directory given by 'runtimepath' and/or 'packpath'.
Ignores non-existing files.

Example:

```
:runtime syntax/c.vim
:runtime syntax/c.lua
```

There can be multiple space-separated {file} arguments. Each {file} is searched for in the first directory from 'runtimepath', then in the second directory, etc.

When [!] is included, all found files are sourced. Else only the first found file is sourced.

When [where] is omitted only 'runtimepath' is used. Other values:

START	search only under "start" in 'packpath'
OPT	search only under "opt" in 'packpath'
PACK	search under "start" and "opt" in 'packpath'
ALL	first use 'runtimepath', then search under "start" and "opt" in 'packpath'

When {file} contains wildcards it is expanded to all matching files. Example:

```
:runtime! plugin/**/*.{vim,lua}
```

This is what Nvim uses to load the plugin files when starting up. This similar command:

```
:runtime plugin/**/*.{vim,lua}
```

would source the first file only.

For each {file} pattern, if two .vim and .lua file names match and differ only in extension, the .vim file is sourced first.

When 'verbose' is one or higher, there is a message when no file could be found.
When 'verbose' is two or higher, there is a message about each searched file.

:pa :packadd E919

:pa[ckadd][!] {name} pack-add Search for an optional plugin directory in 'packpath', source any plugin files found, and add it to 'runtimepath'. The directory must match:

```
pack/*/opt/{name}
```

If the directory pack/*/opt/{name}/after exists it is added at the end of 'runtimepath'.

Note: Use vim.pack.add() to install from a URL.

If loading packages from "pack/*/start" was skipped,
then this directory is searched first:

```
pack/*/start/{name}
```

Note that {name} is the directory name, not the name
of the .vim file. All files matching the patterns

```
pack/*/opt/{name}/plugin/**/*.vim
pack/*/opt/{name}/plugin/**/*.lua
```

will be sourced. This allows for using subdirectories
below "plugin", just like with plugins in
'runtimepath'.

If the filetype detection was already enabled (this
is usually done with a syntax enable or filetype on
command in your vimrc, or automatically during
initialization), and the package was found in
"pack/*/opt/{name}", this command will also look
for "{name}/ftdetect/*.vim" files.

When the optional "!" is given, no plugin/ files or
ftdetect/ scripts are loaded, only the matching
directories are added to 'runtimepath'. This is
useful in your init.vim. The plugins will then be
loaded during the load-plugins initialization step
(note that the loading order will be reversed because
each directory is inserted before others), after
loading the ftdetect scripts.

To programmatically decide if ! is needed during
startup, check v:vim_did_init: use ! if 0 (to not
duplicate load-plugins step), no ! otherwise (to
force load plugin files as otherwise they won't be
loaded automatically).

:packl :packloadall

:packl[oadall][!] Load all packages in the "start" directory under each
entry in 'packpath'.

First all the directories found are added to
'runtimepath', then the plugins found in the
directories are sourced. This allows for a plugin to
depend on something of another plugin, e.g. an
"autoload" directory. See packload-two-steps for
how this can be useful.

This is normally done automatically during startup,
after loading your vimrc file. With this command it
can be done earlier.

Packages will be loaded only once. Using
:packloadall a second time will have no effect.
When the optional ! is added this command will load
packages even when done before.

Note that when using :packloadall in the vimrc
file, the 'runtimepath' option is updated, and later
all plugins in 'runtimepath' will be loaded, which
means they are loaded again. Plugins are expected to
handle that.

An error only causes sourcing the script where it
happens to be aborted, further plugins will be loaded.
See packages.

:scripte[ncoding] [encoding] :scripte :scriptencoding E167
Specify the character encoding used in the script.
The following lines will be converted from [encoding]
to the value of the 'encoding' option, if they are
different. Examples:

```
scriptencoding iso-8859-5
scriptencoding cp932
```

When [encoding] is empty, no conversion is done. This
can be used to restrict conversion to a sequence of
lines:

```
scriptencoding euc-jp
... lines to be converted ...
scriptencoding
... not converted ...
```

When conversion isn't supported by the system, there
is no error message and no conversion is done. When a
line can't be converted there is no error and the
original line is kept.

Don't use "ucs-2" or "ucs-4", scripts cannot be in
these encodings (they would contain NUL bytes).
When a sourced script starts with a BOM (Byte Order
Mark) in utf-8 format Vim will recognize it, no need
to use ":scriptencoding utf-8" then.

:scr :scriptnames

:scr[iptnames] List all sourced script names, in the order they were first sourced. The number is used for the script ID <SID>.
Also see getscriptinfo().

:scr[iptnames][!] {scriptId} :script

Edit script {scriptId}. Although ":scriptnames name" works, using ":script name" is recommended.
When the current buffer can't be abandoned and the ! is not present, the command fails.

:fini :finish E168

:fini[sh] Stop sourcing a script. Can only be used in a Vim script file. This is a quick way to skip the rest of the file. If it is used after a :try but before the matching :finally (if present), the commands following the ":finally" up to the matching :endtry are executed first. This process applies to all nested ":try"s in the script. The outermost ":endtry" then stops sourcing the script.

All commands and command sequences can be repeated by putting them in a named register and then executing it. There are two ways to get the commands in the register:

- Use the record command "q". You type the commands once, and while they are being executed they are stored in a register. Easy, because you can see what you are doing. If you make a mistake, "p"ut the register into the file, edit the command sequence, and then delete it into the register again. You can continue recording by appending to the register (use an uppercase letter).
- Delete or yank the command sequence into the register.

Often used command sequences can be put under a function key with the ':map' command.

An alternative is to put the commands in a file, and execute them with the ':source!' command. Useful for long command sequences. Can be combined with the ':map' command to put complicated commands under a function key.

The ':source' command reads Ex commands from a file or a buffer line by line. You will have to type any needed keyboard input. The ':source!' command reads from a script file character by character, interpreting each character as if you typed it.

Example: When you give the ":!ls" command you get the hit-enter prompt. If you ':source' a file with the line "!ls" in it, you will have to type the <Enter> yourself. But if you ':source!' a file with the line ":!ls" in it, the next characters from that file are read until a <CR> is found. You will not have to type <CR> yourself, unless ":!ls" was the last line in the file.

It is possible to put ':source[!]' commands in the script file, so you can make a top-down hierarchy of script files. The ':source' command can be nested as deep as the number of files that can be opened at one time (about 15). The ':source!' command can be nested up to 15 levels deep.

You can use the "<script>" string (literally, this is not a special key) inside of the sourced file, in places where a file name is expected. It will be replaced by the file name of the sourced file. For example, if you have a "other.vimrc" file in the same directory as your init.vim file, you can source it from your init.vim file with this command:

```
:source <script>:h/other.vimrc
```

In script files terminal-dependent key codes are represented by terminal-independent two character codes. This means that they can be used in the same way on different kinds of terminals. The first character of a key code is 0x80 or 128, shown on the screen as "~@". The second one can be found in the list key-notation. Any of these codes can also be entered with CTRL-V followed by the three digit decimal code.

:source_crnl W15

Windows: Files that are read with ":source" normally have <CR><NL> <EOL>s. These always work. If you are using a file with <NL> <EOL>s (for example, a file made on Unix), this will be recognized if 'fileformats' is not empty and the first line does not end in a <CR>. This fails if the first line has something like ":map <F1> :help^M", where "^M" is a <CR>. If the first line ends in a <CR>, but following ones don't, you will get an error message, because the <CR> from the first lines will be lost.

On other systems, Vim expects ":source"ed files to end in a <NL>. These always work. If you are using a file with <CR><NL> <EOL>s (for example, a file made on MS-Windows), all lines will have a trailing <CR>. This may cause problems for some commands (e.g., mappings). There is no automatic <EOL> detection, because it's common to start with a line that defines a mapping that ends in a <CR>, which will confuse the automaton.

line-continuation

Long lines in a ":source"d Ex command script file can be split by inserting

a line continuation symbol "\" (backslash) at the start of the next line.
There can be white space before the backslash, which is ignored.

Example: the lines

```
:set comments=sr:/*,mb:*,el:*/,
             \://,
             \b:#,
             \:%,
             \n:>,
             \fb:-
```

are interpreted as if they were given in one line:

```
:set comments=sr:/*,mb:*,el:*/,://,b:#,:%,n:>,fb:-
```

All leading whitespace characters in the line before a backslash are ignored.
Note however that trailing whitespace in the line before it cannot be
inserted freely; it depends on the position where a command is split up
whether additional whitespace is allowed or not.

When a space is required it's best to put it right after the backslash. A
space at the end of a line is hard to see and may be accidentally deleted.

```
:syn match Comment
        \ "very long regexp"
        \ keepend
```

There is a problem with the ":append" and ":insert" commands:

```
:1append
\asdf
.
```

The backslash is seen as a line-continuation symbol, thus this results in the
command:

```
:1appendasdf
.
```

To avoid this, add the 'C' flag to the 'cpoptions' option:

```
:set cpo+=C
:1append
\asdf
.
:set cpo-=C
```

Note that when the commands are inside a function, you need to add the 'C'
flag when defining the function, it is not relevant when executing it.

```
:set cpo+=C
:function Foo()
:1append
\asdf
.
:endfunction
:set cpo-=C
```

line-continuation-comment

To add a comment in between the lines start with '"\ '. Notice the space
after the backslash. Example:

```
let array = [
         "\ first entry comment
         \ 'first',
         "\ second entry comment
         \ 'second',
         \ ]
```

Rationale:
        Most programs work with a trailing backslash to indicate line
        continuation. Using this in Vim would cause incompatibility with Vi.
        For example for this Vi mapping:

```
:map xx  asdf\
```

        Therefore the unusual leading backslash is used.

        Starting a comment in a continuation line results in all following
        continuation lines to be part of the comment. Since it was like this
        for a long time, when making it possible to add a comment halfway a
        sequence of continuation lines, it was not possible to use \", since

that was a valid continuation line. Using '"\ ' comes closest, even
though it may look a bit weird. Requiring the space after the
backslash is to make it very unlikely this is a normal comment line.

Debugging scripts

debug-scripts

Besides the obvious messages that you can add to your scripts to find out what they are doing, Vim offers a debug mode. This allows you to step through a sourced file or user function and set breakpoints.

NOTE: The debugging mode is far from perfect. Debugging will have side effects on how Vim works. You cannot use it to debug everything. For example, the display is messed up by the debugging messages.

An alternative to debug mode is setting the 'verbose' option. With a bigger number it will give more verbose messages about what Vim is doing.

STARTING DEBUG MODE

debug-mode

To enter debugging mode use one of these methods:

1. Start Vim with the -D argument:

```
vim -D file.txt
```

   Debugging will start as soon as the first vimrc file is sourced. This is useful to find out what is happening when Vim is starting up. A side effect is that Vim will switch the terminal mode before initialisations have finished, with unpredictable results.
   For a GUI-only version (Windows) the debugging will start as soon as the GUI window has been opened. To make this happen early, add a ":gui" command in the vimrc file.

:debug

2. Run a command with ":debug" prepended. Debugging will only be done while this command executes. Useful for debugging a specific script or user function. And for scripts and functions used by autocommands. Example:

```
:debug edit test.txt.gz
```

3. Set a breakpoint in a sourced file or user function. You could do this in the command line:

```
vim -c "breakadd file */explorer.vim" .
```

   This will run Vim and stop in the first line of the "explorer.vim" script.
   Breakpoints can also be set while in debugging mode.

In debugging mode every executed command is displayed before it is executed. Comment lines, empty lines and lines that are not executed are skipped. When a line contains two commands, separated by "|", each command will be displayed separately.

DEBUG MODE

Once in debugging mode, the usual Ex commands can be used. For example, to inspect the value of a variable:

```
echo idx
```

When inside a user function, this will print the value of the local variable "idx". Prepend "g:" to get the value of a global variable:

```
echo g:idx
```

All commands are executed in the context of the current function or script. You can also set options, for example setting or resetting 'verbose' will show what happens, but you might want to set it just before executing the lines you are interested in:

```
:set verbose=20
```

Commands that require updating the screen should be avoided, because their effect won't be noticed until after leaving debug mode. For example:

```
:help
```

won't be very helpful.

There is a separate command-line history for debug mode.

The line number for a function line is relative to the start of the function. If you have trouble figuring out where you are, edit the file that defines

the function in another Vim, search for the start of the function and do
"99j". Replace "99" with the line number.

Additionally, these commands can be used:

>cont

cont Continue execution until the next breakpoint is hit.

>quit

quit Abort execution. This is like using CTRL-C, some
things might still be executed, doesn't abort
everything. Still stops at the next breakpoint.

>next

next Execute the command and come back to debug mode when
it's finished. This steps over user function calls
and sourced files.

>step

step Execute the command and come back to debug mode for
the next command. This steps into called user
functions and sourced files.

>interrupt

interrupt This is like using CTRL-C, but unlike ">quit" comes
back to debug mode for the next command that is
executed. Useful for testing :finally and :catch
on interrupt exceptions.

>finish

finish Finish the current script or user function and come
back to debug mode for the command after the one that
sourced or called it.

>bt

>backtrace

>where

backtrace Show the call stacktrace for current debugging
session.
bt
where

>frame

frame N Goes to N backtrace level. + and - signs make movement
relative. E.g., ":frame +3" goes three frames up.

>up

up Goes one level up from call stacktrace.

>down

down Goes one level down from call stacktrace.

About the additional commands in debug mode:

- There is no command-line completion for them, you get the completion for the normal Ex commands only.
- You can shorten them, up to a single character, unless more than one command starts with the same letter. "f" stands for "finish", use "fr" for "frame".
- Hitting <CR> will repeat the previous one. When doing another command, this is reset (because it's not clear what you want to repeat).
- When you want to use the Ex command with the same name, prepend a colon: ":cont", ":next", ":finish" (or shorter).

The backtrace shows the hierarchy of function calls, e.g.:

```
>bt
3 function One[3]
2 Two[3]
->1 Three[3]
0 Four
line 1: let four = 4
```

The "->" points to the current frame. Use "up", "down" and "frame N" to
select another frame.

In the current frame you can evaluate the local function variables. There is
no way to see the command at the current line yet.

DEFINING BREAKPOINTS

:breaka :breakadd

:breaka[dd] func [lnum] {name}
Set a breakpoint in a function. Example:

```
:breakadd func Explore
```

Doesn't check for a valid function name, thus the breakpoint can be set before the function is defined.

:breaka[dd] file [lnum] {name}
Set a breakpoint in a sourced file. Example:

```
:breakadd file 43 init.vim
```

:breaka[dd] here
Set a breakpoint in the current line of the current file. Like doing:

```
:breakadd file <cursor-line> <current-file>
```

Note that this only works for commands that are executed when sourcing the file, not for a function defined in that file.

:breaka[dd] expr {expression}
Sets a breakpoint, that will break whenever the {expression} evaluates to a different value. Example:

```
:breakadd expr g:lnum
```

Will break, whenever the global variable lnum changes.

Errors in evaluation are suppressed, you can use the name of a variable that does not exist yet. This also means you will not notice anything if the expression has a mistake.

Note if you watch a script-variable this will break when switching scripts, since the script variable is only valid in the script where it has been defined and if that script is called from several other scripts, this will stop whenever that particular variable will become visible or inaccessible again.

The [lnum] is the line number of the breakpoint. Vim will stop at or after this line. When omitted line 1 is used.

:debug-name

{name} is a pattern that is matched with the file or function name. The pattern is like what is used for autocommands. There must be a full match (as if the pattern starts with "^" and ends in "$"). A "*" matches any sequence of characters. 'ignorecase' is not used, but "\c" can be used in the pattern to ignore case /\c. Don't include the () for the function name!

The match for sourced scripts is done against the full file name. If no path is specified the current directory is used. Examples:

```
breakadd file explorer.vim
```

matches "explorer.vim" in the current directory.

```
breakadd file *explorer.vim
```

matches ".../plugin/explorer.vim", ".../plugin/iexplorer.vim", etc.

```
breakadd file */explorer.vim
```

matches ".../plugin/explorer.vim" and "explorer.vim" in any other directory.

The match for functions is done against the name as it's shown in the output of ":function". For local functions this means that something like "<SNR>99_" is prepended.

Note that functions are first loaded and later executed. When they are loaded the "file" breakpoints are checked, when they are executed the "func" breakpoints.

DELETING BREAKPOINTS

:breakd :breakdel E161

:breakd[el] {nr}
Delete breakpoint {nr}. Use :breaklist to see the number of each breakpoint.

:breakd[el] *
Delete all breakpoints.

:breakd[el] func [lnum] {name}
Delete a breakpoint in a function.

:breakd[el] file [lnum] {name}
Delete a breakpoint in a sourced file.

:breakd[el] here
Delete a breakpoint at the current line of the current file.

When [lnum] is omitted, the first breakpoint in the function or file is deleted.

The {name} must be exactly the same as what was typed for the ":breakadd"
command. "explorer", "*explorer.vim" and "*explorer*" are different.

LISTING BREAKPOINTS

:breakl[ist] :breakl :breaklist
List all breakpoints.

OBSCURE

:debugg[reedy] :debugg :debuggreedy
Read debug mode commands from the normal input stream, instead
of getting them directly from the user. Only useful for test
scripts. Example:

```
echo 'q^Mq' | vim -e -s -c debuggreedy -c 'breakadd file script.vim' -S script.vim
```

:0debugg[reedy]
Undo ":debuggreedy": get debug mode commands directly from the
user, don't use typeahead for debug commands.

Profiling profile profiling

Profiling means that Vim measures the time that is spent on executing
functions and/or scripts.

You can also use the reltime() function to measure time.

For profiling syntax highlighting see :syntime.

For example, to profile the one_script.vim script file:

```
:profile start /tmp/one_script_profile
:profile file one_script.vim
:source one_script.vim
:exit
```

:prof[ile] start {fname} :prof :profile E750
Start profiling, write the output in {fname} upon exit or when
a :profile stop or :profile dump command is invoked.
"~/" and environment variables in {fname} will be expanded.
If {fname} already exists it will be silently overwritten.
The variable v:profiling is set to one.

:prof[ile] stop
Write the collected profiling information to the logfile and
stop profiling. You can use the :profile start command to
clear the profiling statistics and start profiling again.

:prof[ile] pause
Stop profiling until the next :profile continue command.
Can be used when doing something that should not be counted
(e.g., an external command). Does not nest.

:prof[ile] continue
Continue profiling after :profile pause.

:prof[ile] func {pattern}
Profile function that matches the pattern {pattern}.
See :debug-name for how {pattern} is used.

:prof[ile][!] file {pattern}
Profile script file that matches the pattern {pattern}.
See :debug-name for how {pattern} is used.
This only profiles the script itself, not the functions
defined in it.
When the [!] is added then all functions defined in the script
will also be profiled.
Note that profiling only starts when the script is loaded
after this command. A :profile command in the script itself
won't work.

:prof[ile] dump
Write the current state of profiling to the logfile
immediately. After running this command, Vim continues to
collect the profiling statistics.

:profd[el] ... :profd :profdel
Stop profiling for the arguments specified. See :breakdel
for the arguments. Examples:

```
profdel func MyFunc
profdel file MyScript.vim
profdel here
```

You must always start with a ":profile start fname" command. The resulting
file is written when Vim exits. For example, to profile one specific
function:

```
profile start /tmp/vimprofile
profile func MyFunc
```

Here is an example of the output, with line
numbers prepended for the explanation:

```
1 FUNCTION  Test2()
2 Called 1 time
3 Total time:   0.155251
4  Self time:   0.002006
5
6 count  total (s)   self (s)
7       9              0.000096   for i in range(8)
8       8   0.153655   0.000410     call Test3()
9       8              0.000070   endfor
10                                 " Ask a question
11      1              0.001341   echo input("give me an answer: ")
```

The header (lines 1-4) gives the time for the whole function. The "Total"
time is the time passed while the function was executing. The "Self" time is
the "Total" time reduced by time spent in:

- other user defined functions
- sourced scripts
- executed autocommands
- external (shell) commands

Lines 7-11 show the time spent in each executed line. Lines that are not
executed do not count. Thus a comment line is never counted.

The Count column shows how many times a line was executed. Note that the
"for" command in line 7 is executed one more time as the following lines.
That is because the line is also executed to detect the end of the loop.

The time Vim spends waiting for user input isn't counted at all. Thus how
long you take to respond to the input() prompt is irrelevant.

Profiling should give a good indication of where time is spent, but keep in
mind there are various things that may clobber the results:

- Real elapsed time is measured, if other processes are busy they may cause
  delays at unpredictable moments. You may want to run the profiling several
  times and use the lowest results.
- If you have several commands in one line you only get one time. Split the
  line to see the time for the individual commands.
- The time of the lines added up is mostly less than the time of the whole
  function. There is some overhead in between.
- Functions that are deleted before Vim exits will not produce profiling
  information. You can check the v:profiling variable if needed:

```
:if !v:profiling
:   delfunc MyFunc
:endif
```

- Profiling may give weird results on multi-processor systems, when sleep
  mode kicks in or the processor frequency is reduced to save power.
- The "self" time is wrong when a function is used recursively.

Visual mode Visual-mode visual-mode

Visual mode is a flexible and easy way to select a piece of text for an operator. It is the only way to select a block of text (visual-block).

This is introduced in section 04.4 of the user manual.

1. Using Visual mode visual-use

Using Visual mode consists of three parts:

1. Mark the start of the text with "v", "V" or CTRL-V.

   The character under the cursor will be used as the start.
2. Move to the end of the text.

   The text from the start of the Visual mode up to and including the character under the cursor is highlighted.
3. Type an operator command.

   The highlighted characters will be operated upon.

The hl-Visual group determines the highlighting of the visual selection. The 'virtualedit' option can be used to allow positioning the cursor to positions where there is no actual character.

The highlighted text normally includes the character under the cursor. However, when the 'selection' option is set to "exclusive" and the cursor is after the Visual area, the character under the cursor is not included.

With "v" the text before the start position and after the end position will not be highlighted. However, all uppercase and non-alpha operators, except "~" and "U", will work on whole lines anyway. See the list of operators below.

visual-block

With CTRL-V (blockwise Visual mode) the highlighted text will be a rectangle between start position and the cursor. However, some operators work on whole lines anyway (see the list below). The change and substitute operators will delete the highlighted text and then start insertion at the top left position.

2. Starting and stopping Visual mode visual-start

v charwise-visual

[count]v — Start Visual mode per character.
With [count] select the same number of characters or lines as used for the last Visual operation, but at the current cursor position, multiplied by [count].
When the previous Visual operation was on a block both the width and height of the block are multiplied by [count].
When there was no previous Visual operation [count] characters are selected. This is like moving the cursor right N * [count] characters. One less when 'selection' is not "exclusive".

V linewise-visual

[count]V — Start Visual mode linewise.
With [count] select the same number of lines as used for the last Visual operation, but at the current cursor position, multiplied by [count]. When there was no previous Visual operation [count] lines are selected.

CTRL-V blockwise-visual

[count]CTRL-V — Start Visual mode blockwise.

If you use <Esc>, click the left mouse button or use any command that does a jump to another buffer while in Visual mode, the highlighting stops and no text is affected. Also when you hit "v" in charwise Visual mode, "CTRL-V" in blockwise Visual mode or "V" in linewise Visual mode. If you hit CTRL-Z the highlighting stops and the editor is suspended or a new shell is started CTRL-Z.

v_v v_CTRL-V v_V

new mode after typing:	"v"	"CTRL-V"	"V"
old mode			
Normal	Visual	blockwise Visual	linewise Visual
Visual	Normal	blockwise Visual	linewise Visual
blockwise Visual	Visual	Normal	linewise Visual
linewise Visual	Visual	blockwise Visual	Normal

gv v_gv reselect-Visual

gv — Starts Visual mode with the previous Visual area and mode. In Visual mode the current and the previous Visual area are exchanged. After using "p" or "P" in Visual mode the text that was put will be selected.

An operator target that internally creates a temporary Visual selection (gn, "d<C-v>j", an omap-info text object) does not update the previous ("gv") area (nor '< '>, visualmode()).

gn (gn v_gn)
: Searches forward for the last-used search pattern, like with n, and starts Visual mode to select the match. If the cursor is on the match, visually selects it. If an operator is pending, operates on the match. E.g., "dgn" deletes the text of the next match. If Visual mode is active, extends the selection until the end of the next match.

  The Visual marks '< '> are not changed, so "cgn" with a /\%V pattern can be repeated with .

  'wrapscan' applies.
  Note: Unlike n the search direction does not depend on the previous search command.

gN (gN v_gN)
: Like gn but searches backward, like with N.

<LeftMouse>
: Set the current cursor position. If Visual mode is active it is stopped. Only when 'mouse' option contains 'n' or 'a'. If the position is within 'so' lines from the last line on the screen the text is scrolled up. If the position is within 'so' lines from the first line on the screen the text is scrolled down.

<RightMouse>
: Start Visual mode if it is not active. The text from the cursor position to the position of the click is highlighted. If Visual mode was already active move the start or end of the highlighted text, whichever is closest, to the position of the click. Only when 'mouse' option contains 'n' or 'a'.

  Note: when 'mousemodel' is set to "popup", <S-LeftMouse> has to be used instead of <RightMouse>.

<LeftRelease>
: This works like a <LeftMouse>, if it is not at the same position as <LeftMouse>. In an older version of xterm you won't see the selected area until the button is released, unless there is access to the display where the xterm is running (via the DISPLAY environment variable or the -display argument). Only when 'mouse' option contains 'n' or 'a'.

<A-LeftMouse>
: Starts or extends the Visual area blockwise, see <A-LeftMouse>.

If Visual mode is not active and the "v", "V" or CTRL-V is preceded with a count, the size of the previously highlighted area is used for a start. You can then move the end of the highlighted area and give an operator. The type of the old area is used (character, line or blockwise).

- Linewise Visual mode: The number of lines is multiplied with the count.
- Blockwise Visual mode: The number of lines and columns is multiplied with the count.
- Normal Visual mode within one line: The number of characters is multiplied with the count.
- Normal Visual mode with several lines: The number of lines is multiplied with the count, in the last line the same number of characters is used as in the last line in the previously highlighted area.

The start of the text is the Cursor position. If the "$" command was used as one of the last commands to extend the highlighted text, the area will be extended to the rightmost column of the longest line.

If you want to highlight exactly the same area as the last time, you can use "gv" gv v_gv.

<Esc> (v_<Esc>)
: In Visual mode: Stop Visual mode.

  (v_META v_ALT)
  ALT (META) may act like <Esc> if the chord is not mapped. For example <A-x> acts like <Esc>x if <A-x> does not have a visual-mode mapping.

CTRL-C (v_CTRL-C)
: In Visual mode: Stop Visual mode. When insert mode is pending (the mode message shows "-- (insert) VISUAL --"), it is also stopped. On MS-Windows, you may need to press CTRL-Break.

3. Changing the Visual area visual-change

v_o

o Go to Other end of highlighted text: The current cursor position becomes the start of the highlighted text and the cursor is moved to the other end of the highlighted text. The highlighted area remains the same.

v_O

O Go to Other end of highlighted text. This is like "o", but in Visual block mode the cursor moves to the other corner in the same line. When the corner is at a character that occupies more than one position on the screen (e.g., a <Tab>), the highlighted text may change.

v_$

When the "$" command is used with blockwise Visual mode, the right end of the highlighted text will be determined by the longest highlighted line. This stops when a motion command is used that does not move straight up or down.

For moving the end of the block many commands can be used, but you cannot use Ex commands, commands that make changes or abandon the file. Commands (starting with) ".", "&", CTRL-^, "Z", CTRL-], CTRL-T, CTRL-R, CTRL-I and CTRL-O cause a beep and Visual mode continues.

When switching to another window on the same buffer, the cursor position in that window is adjusted, so that the same Visual area is still selected. This is especially useful to view the start of the Visual area in one window, and the end in another. You can then use <RightMouse> (or <S-LeftMouse> when 'mousemodel' is "popup") to drag either end of the Visual area.

4. Operating on the Visual area visual-operators

The operators that can be used are:

~	switch case	v_~
d	delete	v_d
c	change (4)	v_c
y	yank	v_y
>	shift right (4)	v_>
<	shift left (4)	v_<
!	filter through external command (1)	v_!
=	filter through 'equalprg' option command (1)	v_=
gq	format lines to 'textwidth' length (1)	v_gq

The objects that can be used are:

aw	a word (with white space)	v_aw
iw	inner word	v_iw
aW	a WORD (with white space)	v_aW
iW	inner WORD	v_iW
as	a sentence (with white space)	v_as
is	inner sentence	v_is
ap	a paragraph (with white space)	v_ap
ip	inner paragraph	v_ip
al	all lines	v_al
il	inner line	v_il
ab	a () block (with parentheses)	v_ab
ib	inner () block	v_ib
aB	a {} block (with braces)	v_aB
iB	inner {} block	v_iB
at	a <tag> </tag> block (with tags)	v_at
it	inner <tag> </tag> block	v_it
a<	a <> block (with <>)	v_a<
i<	inner <> block	v_i<
a[	a [] block (with [])	v_a[
i[	inner [] block	v_i[
a"	a double quoted string (with quotes)	v_aquote
i"	inner double quoted string	v_iquote
a'	a single quoted string (with quotes)	v_a'
i'	inner simple quoted string	v_i'
a`	a string in backticks (with backticks)	v_a`
i`	inner string in backticks	v_i`

Additionally the following commands can be used:

:	start Ex command for highlighted lines (1)	v_:
r	change (4)	v_r
s	change	v_s
C	change (2)(4)	v_C
S	change (2)	v_S
R	change (2)	v_R
x	delete	v_x
D	delete (3)	v_D
X	delete (2)	v_X
Y	yank (2)	v_Y
p	put	v_p
P	put without overwriting registers	v_P
J	join (1)	v_J
U	make uppercase	v_U

u	make lowercase	v_u
^]	find tag	v_CTRL-]
I	block insert	v_b_I
A	block append	v_b_A

(1): Always whole lines, see :visual example.
(2): Whole lines when not using CTRL-V.
(3): Whole lines when not using CTRL-V, delete until the end of the line when
     using CTRL-V.
(4): When using CTRL-V operates on the block only.

Note that the ":vmap" command can be used to specifically map keys in Visual
mode. For example, if you would like the "/" command not to extend the Visual
area, but instead take the highlighted text and search for that:

```
:vmap / y/<C-R>"<CR>
```

(In the <> notation <>, when typing it you should type it literally; you
need to remove the 'B' flag from 'cpoptions'.)

If you want to give a register name using the """ command, do this just before
typing the operator character: "v{move-around}"xd".

If you want to give a count to the command, do this just before typing the
operator character: "v{move-around}3>" (move lines 3 indents to the right).

{move-around}

The {move-around} is any sequence of movement commands. Note the difference
with {motion}, which is only ONE movement command.

Another way to operate on the Visual area is using the /\%V item in a
pattern. For example, to replace all '(' in the Visual area with '#':

```
:'<,'>s/\%V(/#/g
```

Note that the "'<,'>" will appear automatically when you press ":" in Visual
mode.

5. Blockwise operators blockwise-operators

Reminder: Use 'virtualedit' to be able to select blocks that start or end
after the end of a line or halfway through a tab.

Visual-block Insert v_b_I

With a blockwise selection, I{string}<ESC> will insert {string} at the start
of block on every line of the block, provided that the line extends into the
block. Thus lines that are short will remain unmodified. TABs are split to
retain visual columns. Works only for adding text to a line, not for
deletions. See v_b_I_example.

Visual-block Append v_b_A

With a blockwise selection, A{string}<ESC> will append {string} to the end of
block on every line of the block. There is some differing behavior where the
block RHS is not straight, due to different line lengths:

1. Block was created with <C-v>$

   In this case the string is appended to the end of each line.
2. Block was created with <C-v>{move-around}

   In this case the string is appended to the end of the block on each line,
   and whitespace is inserted to pad to the end-of-block column.

See v_b_A_example.
Note: "I" and "A" behave differently for lines that don't extend into the
selected block. This was done intentionally, so that you can do it the way
you want.
Works only for adding text to a line, not for deletions.

Visual-block change v_b_c

All selected text in the block will be replaced by the same text string. When
using "c" the selected text is deleted and Insert mode started. You can then
enter text (without a line break). When you hit <Esc>, the same string is
inserted in all previously selected lines.

Visual-block Change v_b_C

Like using "c", but the selection is extended until the end of the line for
all lines.

v_b_<

Visual-block Shift v_b_>

The block is shifted by 'shiftwidth'. The RHS of the block is irrelevant.
The LHS of the block determines the point from which to apply a right shift,
and padding includes TABs optimally according to 'ts' and 'et'. The LHS of
the block determines the point up to which to shift left.

See v_b_>_example.
See v_b_<_example.

Visual-block Replace v_b_r

Every screen char in the highlighted region is replaced with the same char, ie
TABs are split and the virtual whitespace is replaced, maintaining screen
layout.
See v_b_r_example.

6. Repeating visual-repeat

Repeating a Visual operation works "semantically": . replays the captured
keysequence (Visual selection plus operator) at the cursor: "viwd" then "."
deletes the word at the cursor, whatever its size. (**Note:** this is unlike Vim,
where Visual-repeat works on the same fixed-size region. If you prefer that
behavior, see visual-fixed-size.)

A [count] given to "." is prepended to the replayed keys. If the selection was
extended by a command that cannot be replayed (mouse, search, Ex command,
scroll), or used gv (e.g. a mapping like ">gv"), "." instead applies the
operator to a same-size area, as if "1v" (v) preceded it.

visual-fixed-size

Example: To mimic Vim's fixed-size Visual dot-repeat: after a Visual
operation, reselect the fixed-size region ("1v", v) and applies the
operator:

```
local vop ---@type string?
vim.api.nvim_create_autocmd('CmdAtom', {
  ---@param ev {data: vim.event.cmdatom.data}
  callback = function(ev)
    if ev.data.type == 'visual' then
      vop = ev.data.atoms and ev.data.atoms[#ev.data.atoms].keys or nil
    elseif ev.data.changed then
      vop = nil  -- the last change is no longer the Visual one
    end
  end,
})
vim.keymap.set('n', '.', function()
  vim.schedule(function()
    vim.api.nvim_feedkeys(vop and ('1v' .. vop) or '.', 'n', false)
  end)
end)
```

The Visual mode default-mappings "@" repeats a register for all selected
lines if the selection is linewise. See v_@-default for details. For
example, given the text:

```
        123(hello)321
        456(world)654
        456(NOT THIS)654
```

With register "x" containing the commands yi(VP, visually selecting the
first two lines and typing @x produces:

```
        hello
        world
        456(NOT THIS)654
```

7. Examples visual-examples

:visual_example

Currently the ":" command works on whole lines only. When you select part of
a line, doing something like ":!date" will replace the whole line. If you
want only part of the line to be replaced you will have to make a mapping for
it. In a future release ":" may work on partial lines.

Here is an example, to replace the selected text with the output of "date":

```
:vmap _a <Esc>`>a<CR><Esc>`<i<CR><Esc>!!date<CR>kgJgJ
```

(In the <> notation <>, when typing it you should type it literally; you
need to remove the 'B' flag from 'cpoptions')

What this does is:

```
<Esc>           stop Visual mode
`>              go to the end of the Visual area
a<CR><Esc>      break the line after the Visual area
`<              jump to the start of the Visual area
i<CR><Esc>      break the line before the Visual area
!!date<CR>      filter the Visual text through date
kJJ             Join the lines back together
```

visual-search

Here is an idea for a mapping that makes it possible to do a search for the
selected text:

```
:vmap X y/<C-R>"<CR>
```

(In the <> notation <>, when typing it you should type it literally; you
need to remove the 'B' flag from 'cpoptions')

Note that special characters (like '.' and "*") will cause problems.

Visual-block Examples blockwise-examples

With the following text, I will indicate the commands to produce the block and
the results below. In all cases, the cursor begins on the 'a' in the first
line of the test text.
The following modeline settings are assumed ":ts=8:sw=4:".

It will be helpful to
:set hls
/<TAB>
where <TAB> is a real TAB. This helps visualise the operations.

The test text is:

```
abcdefghijklmnopqrstuvwxyz
abc             defghijklmnopqrstuvwxyz
abcdef  ghi             jklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
```

1. fo<C-v>3jISTRING<ESC> v_b_I_example

```
abcdefghijklmnSTRINGopqrstuvwxyz
abc           STRING  defghijklmnopqrstuvwxyz
abcdef  ghi   STRING    jklmnopqrstuvwxyz
abcdefghijklmnSTRINGopqrstuvwxyz
```

1. fo<C-v>3j$ASTRING<ESC> v_b_A_example

```
abcdefghijklmnopqrstuvwxyzSTRING
abc             defghijklmnopqrstuvwxyzSTRING
abcdef  ghi             jklmnopqrstuvwxyzSTRING
abcdefghijklmnopqrstuvwxyzSTRING
```

1. fo<C-v>3j3l<.. v_b_<_example

```
abcdefghijklmnopqrstuvwxyz
abc           defghijklmnopqrstuvwxyz
abcdef  ghi   jklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
```

1. fo<C-v>3j>.. v_b_>_example

```
abcdefghijklmn            opqrstuvwxyz
abc                         defghijklmnopqrstuvwxyz
abcdef  ghi                         jklmnopqrstuvwxyz
abcdefghijklmn            opqrstuvwxyz
```

1. fo<C-v>5l3jrX v_b_r_example

```
abcdefghijklmnXXXXXXuvwxyz
abc           XXXXXXhijklmnopqrstuvwxyz
abcdef  ghi   XXXXXX    jklmnopqrstuvwxyz
abcdefghijklmnXXXXXXuvwxyz
```

8. Select mode Select Select-mode

Select mode looks like Visual mode, but the commands accepted are quite
different. This resembles the selection mode in Microsoft Windows.
When the 'showmode' option is set, "-- SELECT --" is shown in the last line.

Entering Select mode:

- Using the mouse to select an area, and 'selectmode' contains "mouse".

  'mouse' must also contain a flag for the current mode.
- Using a non-printable movement command, with the Shift key pressed, and

  'selectmode' contains "key". For example: <S-Left> and <S-End>. 'keymodel'
  must also contain "startsel".
- Using "v", "V" or CTRL-V command, and 'selectmode' contains "cmd".

- Using "gh", "gH" or "g_CTRL-H" command in Normal mode.

- From Visual mode, press CTRL-G. v_CTRL-G

Commands in Select mode:

- Printable characters, <NL> and <CR> cause the selection to be deleted, and
  Vim enters Insert mode. The typed character is inserted.
- Non-printable movement commands, with the Shift key pressed, extend the
  selection. 'keymodel' must include "startsel".
- Non-printable movement commands, with the Shift key NOT pressed, stop Select
  mode. 'keymodel' must include "stopsel".
- ESC stops Select mode.
- CTRL-O switches to Visual mode for the duration of one command. v_CTRL-O
- CTRL-G switches to Visual mode.
- CTRL-R {register} selects the register to be used for the text that is
  deleted when typing text. v_CTRL-R
  Unless you specify the "_" (black hole) register, the unnamed register is
  also overwritten.

Otherwise, typed characters are handled as in Visual mode.

When using an operator in Select mode, and the selection is linewise, the selected lines are operated upon, but like in charwise selection. For example, when a whole line is deleted, it can later be pasted in the middle of a line.

Mappings and menus in Select mode. Select-mode-mapping

When mappings and menus are defined with the :vmap or :vmenu command they work both in Visual mode and in Select mode. When these are used in Select mode Vim automatically switches to Visual mode, so that the same behavior as in Visual mode is effective. If you don't want this use :xmap or :smap.

One particular edge case:

```
:vnoremap <C-K> <Esc>
```

This ends Visual mode when in Visual mode, but in Select mode it does not work, because Select mode is restored after executing the mapped keys. You need to use:

```
:snoremap <C-K> <Esc>
```

Users will expect printable characters to replace the selected area. Therefore avoid mapping printable characters in Select mode. Or use :sunmap after :map and :vmap to remove it for Select mode.

After the mapping or menu finishes, the selection is enabled again and Select mode entered, unless the selected area was deleted, another buffer became the current one or the window layout was changed.

When a character was typed that causes the selection to be deleted and Insert mode started, Insert mode mappings are applied to this character. This may cause some confusion, because it means Insert mode mappings apply to a character typed in Select mode. Language mappings apply as well.

gV v_gV

gV — Avoid the automatic reselection of the Visual area after a Select mode mapping or menu has finished. Put this just before the end of the mapping or menu. At least it should be after any operations on the selection.

gh

gh — Start Select mode, charwise. This is like "v", but starts Select mode instead of Visual mode. Mnemonic: "get highlighted".

gH

gH — Start Select mode, linewise. This is like "V", but starts Select mode instead of Visual mode. Mnemonic: "get Highlighted".

g_CTRL-H

g CTRL-H — Start Select mode, blockwise. This is like CTRL-V, but starts Select mode instead of Visual mode. Mnemonic: "get Highlighted".

Predefined variables v:var v:

Most variables are read-only, when a variable can be set by the user, it will
be mentioned at the variable description below. The type cannot be changed.

v:argf argf-variable

v:argf
File arguments (expanded to absolute paths) given at startup.

Unlike v:argv, this does not include option arguments such as -u, --cmd, or +cmd. Unlike argv(), it is not affected by later :args, :argadd, or plugin modifications.

Example:

```
nvim file1.txt +ls -- file2.txt
:echo v:argf
" ['/path/to/cwd/file1.txt', '/path/to/cwd/file2.txt']
```

v:argv argv-variable

v:argv
Command line arguments (-u, --cmd, +cmd, …) Nvim was invoked with. The first item is the Nvim command.

See v:progpath to get the full path to Nvim.
See v:argf to get only file args, without other options.

v:cascading cascading-variable

v:cascading
v:true during multicursor cascade, v:false otherwise.
Read-only.

v:char char-variable

v:char
Argument for evaluating 'formatexpr' and used for the typed character when using <expr> in an abbreviation :map-<expr>.
It is also used by the InsertCharPre, InsertEnter, CmdlineLeave and CmdlineLeavePre events.

v:charconvert_from charconvert_from-variable

v:charconvert_from
The name of the character encoding of a file to be converted.
Only valid while evaluating the 'charconvert' option.

v:charconvert_to charconvert_to-variable

v:charconvert_to
The name of the character encoding of a file after conversion.
Only valid while evaluating the 'charconvert' option.

v:cmdarg cmdarg-variable

v:cmdarg
The extra arguments ("++p", "++enc=", "++ff=") given to a file read/write command. This is set before an autocommand event for a file read/write command is triggered. There is a leading space to make it possible to append this variable directly after the read/write command. **Note:** "+cmd" isn't included here, because it will be executed anyway.

v:cmdbang cmdbang-variable

v:cmdbang
Set like v:cmdarg for a file read/write command. When a "!" was used the value is 1, otherwise it is 0. Note that this can only be used in autocommands. For user commands <bang> can be used.

v:collate collate-variable

v:collate
The current locale setting for collation order of the runtime environment. This allows Vim scripts to be aware of the current locale encoding. Technical: it's the value of LC_COLLATE. When not using a locale the value is "C".
This variable can not be set directly, use the :language command.
See multi-lang.

v:completed_item completed_item-variable

v:completed_item
Dictionary containing the complete-items for the most recently completed word after CompleteDone. Empty if the completion failed, or after leaving and re-entering insert mode.
Note: Plugins can modify the value to emulate the builtin CompleteDone event behavior.

v:count count-variable

v:count
The count given for the last Normal mode command. Can be used to get the count before a mapping. Read-only. Example:

```
:map _x :<C-U>echo "the count is " .. v:count<CR>
```

Note: The <C-U> is required to remove the line range that you get when typing ':' after a count.
When there are two counts, as in "3d2w", they are multiplied, just like what happens in the command, "d6w" for the example.
Also used for evaluating the 'formatexpr' option.

v:count1 count1-variable

v:count1
Just like "v:count", but defaults to one when no count is used.

v:ctype ctype-variable

v:ctype
The current locale setting for characters of the runtime environment. This allows Vim scripts to be aware of the current locale encoding. Technical: it's the value of LC_CTYPE. When not using a locale the value is "C". This variable can not be set directly, use the :language command.
See multi-lang.

v:dying dying-variable

v:dying
Normally zero. When a deadly signal is caught it's set to one. When multiple signals are caught the number increases. Can be used in an autocommand to check if Vim didn't terminate normally.
Example:

```
:au VimLeave * if v:dying | echo "\nAAAAaaaarrrggghhhh!!!\n" | endif
```

Note: if another deadly signal is caught when v:dying is one, VimLeave autocommands will not be executed.

v:echospace echospace-variable

v:echospace
Number of screen cells that can be used for an :echo message in the last screen line before causing the hit-enter prompt (no longer applicable when ui2 is enabled).

Depends on 'showcmd', 'ruler' and 'columns'. You need to check 'cmdheight' for whether there are full-width lines available above the last line.

v:errmsg errmsg-variable

v:errmsg
Last error message that occurred (not necessarily displayed). Modifiable (can be set).
Example:

```
let v:errmsg = ""
silent! next
if v:errmsg != ""
  " ... handle error
```

v:errors errors-variable assert-return

v:errors
Errors found by assert functions, such as assert_true().
This is a list of strings.
The assert functions append an item when an assert fails.
The return value indicates this: a one is returned if an item was added to v:errors, otherwise zero is returned.
To remove old results make it empty:

```
let v:errors = []
```

If v:errors is set to anything but a list it is made an empty list by the assert function.

v:event event-variable

v:event
Dictionary of event data for the current autocommand. Valid only during the event lifetime; storing or passing v:event is invalid! Copy it instead:

```
au TextYankPost * let g:foo = deepcopy(v:event)
```

Keys vary by event; see the documentation for the specific event, e.g. DirChanged or TextYankPost.

KEY	DESCRIPTION
abort	Whether the event triggered during an aborting condition (e.g. c_Esc or c_CTRL-C for CmdlineLeave).
chan	channel-id
changed_window	Is v:true if the event fired while changing window (or tab) on DirChanged.
cmdlevel	Level of cmdline.
cmdtype	Type of cmdline, cmdline-char.
col	Column count of popup menu on CompleteChanged, relative to screen.
complete_type	See complete_info_mode
complete_word	The selected word, or empty if completion was abandoned/discarded.
completed_item	Current selected item on CompleteChanged, or {} if no item selected.
cwd	Current working directory.
height	Height of popup menu on CompleteChanged
inclusive	Motion is inclusive, else exclusive.
info	Dict of arbitrary event data.

operator Current operator. Also set for Ex commands (unlike v:operator). For example if TextYankPost is triggered by the :yank Ex command then v:event.operator is "y".
reason CompleteDone reason.
regcontents Text stored in the register as a readfile()-style list of lines.
regname Requested register (e.g "x" for "xyy), or empty string for an unnamed operation.
regtype Type of register as returned by getregtype().
row Row count of popup menu on CompleteChanged, relative to screen.
scope Event-specific scope name.
scrollbar v:true if popup menu has a scrollbar, or v:false if not.
size Total number of completion items on CompleteChanged.
status Job status or exit code, -1 means "unknown". TermClose
visual Selection is visual (as opposed to e.g. a motion range).
width Width of popup menu on CompleteChanged
windows List of window IDs that changed on WinResized

v:exception exception-variable

v:exception

The value of the exception most recently caught and not finished. See also v:stacktrace, v:throwpoint, and throw-variables.
Example:

```
try
  throw "oops"
catch /.*/
  echo "caught " .. v:exception
endtry
```

Output: "caught oops".

v:exiting exiting-variable

v:exiting

Exit code, or v:null before invoking the VimLeavePre and VimLeave autocmds. See :q, :x and :cquit.
Example:

```
:au VimLeave * echo "Exit code is " .. v:exiting
```

v:exitreason exitreason-variable

v:exitreason

Reason for the current exit. Set before QuitPre. Reset if exit was canceled.

Possible values:

- "" Not exiting, or exit was canceled.
- "quit" :quit, :qall, :wq, ZZ, ZQ, etc.
- "restart" :restart, ZR.
- "restart!" :restart!, [count]ZR.

Example:

```
autocmd ExitPre * if v:exitreason ==# 'restart' | echomsg 'restarting' | endif
```

Read-only.

v:false false-variable

v:false

Special value used to put "false" in JSON and msgpack. See json_encode(). This value is converted to "v:false" when used as a String (e.g. in expr5 with string concatenation operator) and to zero when used as a Number (e.g. in expr5 or expr7 when used with numeric operators). Read-only.

v:fcs_choice fcs_choice-variable

v:fcs_choice

What should happen after a FileChangedShell event was triggered. Can be used in an autocommand to tell Vim what to do with the affected buffer:
reload Reload the buffer (does not work if the file was deleted).
edit Reload the buffer and detect the values for options such as 'fileformat', 'fileencoding', 'binary' (does not work if the file was deleted).
ask Ask the user what to do, as if there

was no autocommand. Except that when
only the timestamp changed nothing
will happen.
<empty> Nothing, the autocommand should do
everything that needs to be done.
The default is empty. If another (invalid) value is used then
Vim behaves like it is empty, there is no warning message.

v:fcs_reason fcs_reason-variable

v:fcs_reason

The reason why the FileChangedShell event was triggered.
Can be used in an autocommand to decide what to do and/or what
to set v:fcs_choice to. Possible values:

- deleted file no longer exists
- conflict file contents, mode or timestamp was changed and buffer is modified
- changed file contents has changed
- mode mode of file changed
- time only file timestamp changed

v:fname fname-variable

v:fname

When evaluating 'includeexpr': the file name that was
detected. Empty otherwise.

v:fname_diff fname_diff-variable

v:fname_diff

The name of the diff (patch) file. Only valid while
evaluating 'patchexpr'.

v:fname_in fname_in-variable

v:fname_in

The name of the input file. Valid while evaluating:

option	used for
'charconvert'	file to be converted
'diffexpr'	original file
'patchexpr'	original file

And set to the swap file name for SwapExists.

v:fname_new fname_new-variable

v:fname_new

The name of the new version of the file. Only valid while
evaluating 'diffexpr'.

v:fname_out fname_out-variable

v:fname_out

The name of the output file. Only valid while
evaluating:

option	used for
'charconvert'	resulting converted file [1]
'diffexpr'	output of diff
'patchexpr'	resulting patched file

[1] When doing conversion for a write command (e.g., ":w
file") it will be equal to v:fname_in. When doing conversion
for a read command (e.g., ":e file") it will be a temporary
file and different from v:fname_in.

v:folddashes folddashes-variable

v:folddashes

Used for 'foldtext': dashes representing foldlevel of a closed
fold.
Read-only in the sandbox. fold-foldtext

v:foldend foldend-variable

v:foldend

Used for 'foldtext': last line of closed fold.
Read-only in the sandbox. fold-foldtext

v:foldlevel foldlevel-variable

v:foldlevel

Used for 'foldtext': foldlevel of closed fold.
Read-only in the sandbox. fold-foldtext

v:foldstart foldstart-variable

v:foldstart

Used for 'foldtext': first line of closed fold.
Read-only in the sandbox. fold-foldtext

v:hlsearch hlsearch-variable

v:hlsearch

Variable that indicates whether search highlighting is on.
Setting it makes sense only if 'hlsearch' is enabled. Setting
this variable to zero acts like the :nohlsearch command,
setting it to one acts like

```
let &hlsearch = &hlsearch
```

Note that the value is restored when returning from a
function. function-search-undo.

v:insertmode insertmode-variable

v:insertmode
Used for the InsertEnter and InsertChange autocommand
events. Values:
i Insert mode
r Replace mode
v Virtual Replace mode

v:key key-variable

v:key
Key of the current item of a Dictionary. Only valid while
evaluating the expression used with map() and filter().
Read-only.

v:lang lang-variable

v:lang
The current locale setting for messages of the runtime
environment. This allows Vim scripts to be aware of the
current language. Technical: it's the value of LC_MESSAGES.
The value is system dependent.
This variable can not be set directly, use the :language
command.
It can be different from v:ctype when messages are desired
in a different language than what is used for character
encoding. See multi-lang.

v:lc_time lc_time-variable

v:lc_time
The current locale setting for time messages of the runtime
environment. This allows Vim scripts to be aware of the
current language. Technical: it's the value of LC_TIME.
This variable can not be set directly, use the :language
command. See multi-lang.

v:lnum lnum-variable

v:lnum
Line number for the 'foldexpr' fold-expr, 'formatexpr',
'indentexpr' and 'statuscolumn' expressions, tab page number
for 'guitablabel' and 'guitabtooltip'. Only valid while one of
these expressions is being evaluated. Read-only when in the
sandbox.

v:lua lua-variable

v:lua
Prefix for calling Lua functions from expressions.
See v:lua-call for more information.

v:maxcol maxcol-variable

v:maxcol
Maximum line length. Depending on where it is used it can be
screen columns, characters or bytes. The value currently is
2147483647 on all systems.

v:mouse_col mouse_col-variable

v:mouse_col
Column number for a mouse click obtained with getchar().
This is the screen column number, like with virtcol(). The
value is zero when there was no mouse button click.

v:mouse_lnum mouse_lnum-variable

v:mouse_lnum
Line number for a mouse click obtained with getchar().
This is the text line number, not the screen line number. The
value is zero when there was no mouse button click.

v:mouse_win mouse_win-variable

v:mouse_win
Window number for a mouse click obtained with getchar().
First window has number 1, like with winnr(). The value is
zero when there was no mouse button click.

v:mouse_winid mouse_winid-variable

v:mouse_winid
window-ID for a mouse click obtained with getchar().
The value is zero when there was no mouse button click.

v:msgpack_types msgpack_types-variable

v:msgpack_types
Dictionary containing msgpack types used by msgpackparse()
and msgpackdump(). All types inside dictionary are fixed
(not editable) empty lists. To check whether some list is one
of msgpack types, use is operator.

v:null null-variable

v:null
Special value used to put "null" in JSON and NIL in msgpack.
See json_encode(). This value is converted to "v:null" when
used as a String (e.g. in expr5 with string concatenation
operator) and to zero when used as a Number (e.g. in expr5
or expr7 when used with numeric operators). Read-only.
In some places v:null can be used for a List, Dict, etc.
that is not set. That is slightly different than an empty
List, Dict, etc.

v:numbermax numbermax-variable
v:numbermax Maximum value of a number.

v:numbermin numbermin-variable
v:numbermin Minimum value of a number (negative).

v:numbersize numbersize-variable
v:numbersize
Number of bits in a Number. This is normally 64, but on some systems it may be 32.

v:oldfiles oldfiles-variable
v:oldfiles
List of file names that is loaded from the shada file on startup. These are the files that Vim remembers marks for. The length of the List is limited by the ' argument of the 'shada' option (default is 100).
When the shada file is not used the List is empty.
Also see :oldfiles and c_#<.
The List can be modified, but this has no effect on what is stored in the shada file later. If you use values other than String this will cause trouble.

v:operator operator-variable
v:operator
The last operator given in Normal mode. This is a single character except for commands starting with <g> or <z>, in which case it is two characters. Best used alongside v:prevcount and v:register. Useful if you want to cancel Operator-pending mode and then use the operator, e.g.:

```
:omap O <Esc>:call MyMotion(v:operator)<CR>
```

The value remains set until another operator is entered, thus don't expect it to be empty.
v:operator is not set for :delete, :yank or other Ex commands.
Read-only.

v:option_command option_command-variable
v:option_command
Command used to set the option. Valid while executing an OptionSet autocommand.

value	option was set via
"setlocal"	:setlocal or :let l:xxx
"setglobal"	:setglobal or :let g:xxx
"set"	:set or :let
"modeline"	modeline

v:option_new option_new-variable
v:option_new
New value of the option. Valid while executing an OptionSet autocommand.

v:option_old option_old-variable
v:option_old
Old value of the option. Valid while executing an OptionSet autocommand. Depending on the command used for setting and the kind of option this is either the local old value or the global old value.

v:option_oldglobal option_oldglobal-variable
v:option_oldglobal
Old global value of the option. Valid while executing an OptionSet autocommand.

v:option_oldlocal option_oldlocal-variable
v:option_oldlocal
Old local value of the option. Valid while executing an OptionSet autocommand.

v:option_type option_type-variable
v:option_type
Scope of the set command. Valid while executing an OptionSet autocommand. Can be either "global" or "local"

v:prevcount prevcount-variable
v:prevcount
The count given for the last but one Normal mode command. This is the v:count value of the previous command. Useful if you want to cancel Visual or Operator-pending mode and then use the count, e.g.:

```
:vmap % <Esc>:call MyFilter(v:prevcount)<CR>
```

Read-only.

v:profiling profiling-variable
v:profiling
Normally zero. Set to one after using ":profile start".
See profiling.

v:progname progname-variable
v:progname
The name by which Nvim was invoked (with path removed).
Read-only.

v:progpath progpath-variable

v:progpath
Absolute path to the current running Nvim.
Read-only.

v:register register-variable

v:register
The name of the register in effect for the current normal mode command (regardless of whether that command actually used a register). Or for the currently executing normal mode mapping (use this in custom commands that take a register).
If none is supplied it is the default register '"', unless 'clipboard' contains "unnamed" or "unnamedplus", then it is "*" or '+'.
Also see getreg() and setreg()

v:relnum relnum-variable

v:relnum
Relative line number for the 'statuscolumn' expression.
Read-only.

v:scrollstart scrollstart-variable

v:scrollstart
String describing the script or function that caused the screen to scroll up. It's only set when it is empty, thus the first reason is remembered. It is set to "Unknown" for a typed command.

v:searchforward searchforward-variable

v:searchforward
Search direction: 1 after a forward search, 0 after a backward search. It is reset to forward when directly setting the last search pattern, see quote/.
Note that the value is restored when returning from a function. function-search-undo.
Read-write.

v:servername servername-variable

v:servername
Primary listen-address of Nvim, the first item returned by serverlist(). Usually this is the named pipe created by Nvim at startup or given by --listen (or the deprecated $NVIM_LISTEN_ADDRESS env var).

See also serverstart() serverstop().
Read-only.

$NVIM

$NVIM is set to v:servername by terminal and jobstart(), and is thus a hint that the current environment is a child (direct subprocess) of Nvim.

Example: a child Nvim process can detect and make requests to its parent Nvim:

```
if vim.env.NVIM then
  local ok, chan = pcall(vim.fn.sockconnect, 'pipe', vim.env.NVIM, {rpc=true})
  if ok and chan then
    local client = vim.api.nvim_get_chan_info(chan).client
    local rv = vim.rpcrequest(chan, 'nvim_exec_lua', [[return ... + 1]], { 41 })
    vim.print(('got "%s" from parent Nvim'):format(rv))
  end
end
```

v:shell_error shell_error-variable

v:shell_error
Result of the last shell command. When non-zero, the last shell command had an error. When zero, there was no problem. This only works when the shell returns the error code to Vim. The value -1 is often used when the command could not be executed. Read-only.
Example:

```
!mv foo bar
if v:shell_error
  echo 'could not rename "foo" to "bar"!'
endif
```

v:stacktrace stacktrace-variable

v:stacktrace
The stack trace of the exception most recently caught and not finished. Refer to getstacktrace() for the structure of stack trace. See also v:exception, v:throwpoint, and throw-variables.

v:startreason startreason-variable

v:startreason
The reason Nvim started. Possible values:

- "normal" Normal startup, yearning for life, etc.
- "restart" Started by :restart.

- "restart!" Started by :restart!.

Read-only.

v:starttime starttime-variable

v:starttime
Timestamp (nanoseconds since UNIX epoch) when the Nvim process started.

To see the current "uptime":

```
vim.print(('uptime: %d seconds'):format(os.time() - (vim.v.starttime / 1e9)))
```

Read-only.

v:statusmsg statusmsg-variable

v:statusmsg
Last given status message.
Modifiable (can be set).

v:stderr stderr-variable

v:stderr
channel-id corresponding to stderr. The value is always 2; use this variable to make your code more descriptive. Unlike stdin and stdout (see stdioopen()), stderr is always open for writing. Example:

```
:call chansend(v:stderr, "error: toaster empty\n")
```

v:swapchoice swapchoice-variable

v:swapchoice
SwapExists autocommands can set this to the selected choice for handling an existing swapfile:

```
'o'    Open read-only
'e'    Edit anyway
'r'    Recover
'd'    Delete swapfile
'q'    Quit
'a'    Abort
```

The value should be a single-character string. An empty value results in the user being asked, as would happen when there is no SwapExists autocommand. The default is empty.

v:swapcommand swapcommand-variable

v:swapcommand
Normal mode command to be executed after a file has been opened. Can be used for a SwapExists autocommand to have another Vim open the file and jump to the right place. For example, when jumping to a tag the value is ":tag tagname\r". For ":edit +cmd file" the value is ":cmd\r".

v:swapname swapname-variable

v:swapname
Name of the swapfile found.
Only valid during SwapExists event.
Read-only.

v:t_blob t_blob-variable v:t_TYPE

v:t_blob Value of Blob type. Read-only. See: type()

v:t_bool t_bool-variable

v:t_bool Value of Boolean type. Read-only. See: type()

v:t_dict t_dict-variable

v:t_dict Value of Dictionary type. Read-only. See: type()

v:t_float t_float-variable

v:t_float Value of Float type. Read-only. See: type()

v:t_func t_func-variable

v:t_func Value of Funcref type. Read-only. See: type()

v:t_list t_list-variable

v:t_list Value of List type. Read-only. See: type()

v:t_number t_number-variable

v:t_number Value of Number type. Read-only. See: type()

v:t_string t_string-variable

v:t_string Value of String type. Read-only. See: type()

v:termrequest termrequest-variable

v:termrequest
The value of the most recent OSC, DCS or APC control sequence sent from a process running in the embedded terminal. This can be read in a TermRequest event handler to respond to queries from embedded applications.

v:termresponse termresponse-variable

v:termresponse
The most recent OSC, DCS, APC, or recognized CSI (DA1, kitty multiple-cursors) control sequence received by Nvim from the host-terminal. Can be read in a TermResponse event handler after querying the terminal.

v:testing testing-variable

v:testing Must be set before using test_garbagecollect_now().

v:this_session this_session-variable
v:this_session
Full filename of the last loaded or saved session file.
Empty when no session file has been saved. See :mksession.
Modifiable (can be set).

v:throwpoint throwpoint-variable
v:throwpoint
The point where the exception most recently caught and not
finished was thrown. Not set when commands are typed. See
also v:exception, v:stacktrace, and throw-variables.
Example:

```
try
  throw "oops"
catch /.*/
  echo "Exception from" v:throwpoint
endtry
```

Output: "Exception from test.vim, line 2"

v:true true-variable
v:true
Special value used to put "true" in JSON and msgpack. See
json_encode(). This value is converted to "v:true" when used
as a String (e.g. in expr5 with string concatenation
operator) and to one when used as a Number (e.g. in expr5 or
expr7 when used with numeric operators). Read-only.

v:useractive useractive-variable user-idle
v:useractive
Timestamp (nanoseconds since UNIX epoch) indicating the most
recent user activity, i.e. when a key is received from a UI
(TUI input or nvim_input()).

Initialized to 0 (no user activity since startup).
Read-only.

v:val val-variable
v:val
Value of the current item of a List or Dictionary. Only
valid while evaluating the expression used with map() and
filter(). Read-only.

v:version version-variable
v:version
Vim version number: major version times 100 plus minor
version. Vim 5.0 is 500, Vim 5.1 is 501.
Read-only.
Use has() to check the Nvim (not Vim) version:

```
:if has("nvim-0.2.1")
```

v:versionlong versionlong-variable
v:versionlong
Like v:version, but also including the patchlevel in the last
four digits. Version 8.1 with patch 123 has value 8010123.
This can be used like this:

```
if v:versionlong >= 8010123
```

However, if there are gaps in the list of patches included
this will not work well. This can happen if a recent patch
was included into an older version, e.g. for a security fix.
Use the has() function to make sure the patch is actually
included.

v:vim_did_enter vim_did_enter-variable
v:vim_did_enter
0 during startup, 1 just before VimEnter.
See also v:vim_did_init, which is set earlier.
Read-only.

v:vim_did_init vim_did_init-variable
v:vim_did_init
0 during initialization, 1 after sourcing the user vimrc,
just before load-plugins.
See also v:vim_did_enter, which is set later.
Read-only.

v:virtnum virtnum-variable
v:virtnum
Virtual line number for the 'statuscolumn' expression.
Negative when drawing the status column for virtual lines, zero
when drawing an actual buffer line, and positive when drawing
the wrapped part of a buffer line.
Read-only.

v:warningmsg warningmsg-variable
v:warningmsg
Last given warning message.
Modifiable (can be set).

v:windowid windowid-variable
v:windowid
Application-specific window "handle" which may be set by any

attached UI. Defaults to zero.
Note: For Nvim windows use winnr() or win_getid(), see
window-ID.

Editing with multiple windows and buffers. buffers

The commands which have been added to use multiple windows and buffers are explained here. Additionally, there are explanations for commands that work differently when used in combination with more than one window.

The basics are explained in chapter 7 and 8 of the user manual usr_07.txt usr_08.txt.

1. Introduction windows-intro window

Summary:
A buffer is the in-memory text of a file.
A window is a viewport on a buffer.
A tabpage is a collection of windows.

A window is a viewport onto a buffer. You can use multiple windows on one buffer, or several windows on different buffers.

A buffer is a file loaded into memory for editing. The original file remains unchanged until you write the buffer to the file.

A buffer can be in one of three states:

active-buffer

active: The buffer is displayed in a window. If there is a file for this buffer, it has been read into the buffer. The buffer may have been modified since then and thus be different from the file.

hidden-buffer

hidden: The buffer is not displayed. If there is a file for this buffer, it has been read into the buffer. Otherwise it's the same as an active buffer, you just can't see it.

inactive-buffer

inactive: The buffer is not displayed and does not contain anything. Options for the buffer are remembered if the file was once loaded. It can contain marks from the shada file. But the buffer doesn't contain text.

In a table:

state	displayed	loaded	":buffers"
in window shows			
active	yes	yes	'a'
hidden	no	yes	'h'
inactive	no	no	' '

buffer-reuse

Each buffer has a unique number and the number will not change within a Vim session. The bufnr() and bufname() functions can be used to convert between a buffer name and the buffer number. There is one exception: if a new empty buffer is created and it is not modified, the buffer will be re-used when loading another file into that buffer. This also means the buffer number will not change.

The main Vim window can hold several split windows. There are also tabpages, each of which can hold multiple windows.

focusable

If a window is focusable, it is part of the "navigation stack", that is, editor commands such as :windo, CTRL-W, etc., will consider the window as one that can be made the "current window". A non-focusable window will be skipped by such commands as it isn't assigned a window number. It can be explicitly focused by nvim_set_current_win(), because it is still assigned a window-ID. If it is focused, it will also have a window number. Non-focusable windows are not listed by :tabs, or counted by the default 'tabline'. Their buffer content is not included in 'complete' "w" completion.

Windows (especially floating windows) can have many other api-win_config properties such as "hide" and "fixed" which also affect behavior.

window-hidden

If the api-win_config hide flag is set, then the window is "hidden": it exists but is not shown anywhere. Hidden windows may be focused (the cursor may enter it), so the cursor can perform actions in it, but the cursor will be invisible. Hidden windows and their statuslines are not redrawn.

window-ID winid windowid

Each window has a unique identifier called the window ID, which is permanent during the Nvim session. The win_getid() and win_id2tabwin() functions convert between the window/tab "number" and the identifier.

window-number

Windows are numbered according to their arrangement in a tabpage; this window "number" is given by winnr() and may change whenever windows are opened or closed. The window number is only valid in one specific tab, whereas the window ID is valid globally, across tabs. For most functions that take a window ID or a window number, the window number only applies to the current tab, while the window ID can refer to a window in any tab.

2. Starting Vim windows-starting

By default, Vim starts with one window, just like Vi.

The "-o" and "-O" arguments to Vim can be used to open a window for each file in the argument list. The "-o" argument will split the windows horizontally; the "-O" argument will split the windows vertically. If both "-o" and "-O" are given, the last one encountered will be used to determine the split orientation. For example, this will open three windows, split horizontally:

```
vim -o file1 file2 file3
```

"-oN", where N is a decimal number, opens N windows split horizontally. If there are more file names than windows, only N windows are opened and some files do not get a window. If there are more windows than file names, the last few windows will be editing empty buffers. Similarly, "-ON" opens N windows split vertically, with the same restrictions.

If there are many file names, the windows will become very small. You might want to set the 'winheight' and/or 'winwidth' options to create a workable situation.

Buf/Win Enter/Leave autocommands are not executed when opening the new windows and reading the files, that's only done when they are really entered.

status-line

A status line will be used to separate windows. The 'laststatus' option tells when the last window also has a status line:

'laststatus' = 0	never a status line
'laststatus' = 1	status line if there is more than one window
'laststatus' = 2	always a status line
'laststatus' = 3	have a global statusline at the bottom instead of one for each window

You can change the contents of the status line with the 'statusline' option. This option can be local to the window, so that you can have a different status line in each window.

Normally, inversion is used to display the status line. This can be changed with the hl-StatusLine highlight group. If no highlighting is used for the status line, the '^' character is used for the current window, and '=' for other windows. If 'mouse' is enabled, a status line can be dragged to resize windows.

filler-lines

The lines after the last buffer line in a window are called filler lines. By default, these lines start with a tilde (~) character. The "eob" item in the 'fillchars' option can be used to change this character. By default, these characters are highlighted as NonText (hl-NonText). The EndOfBuffer highlight group (hl-EndOfBuffer) can be used to change the highlighting of the filler characters.

3. Opening and closing a window opening-window

CTRL-W s CTRL-W_s

CTRL-W S CTRL-W_S

CTRL-W CTRL-S CTRL-W_CTRL-S

:[N]sp[lit] [++opt] [+cmd] :sp :split

Split current window in two. The result is two viewports on the same file.

Make the new window N high (default is to use half the height of the current window). Reduces the current window height to create room (and others, if the 'equalalways' option is set, 'eadirection' isn't "hor", and one of them is higher than the current or the new window).

Note: CTRL-S does not work on all terminals and might block further input, use CTRL-Q to get going again.
Also see ++opt and +cmd.

E242 E1159

Be careful when splitting a window in an autocommand, it may mess up the window layout if this happens while making other window layout changes.

:[N]sp[lit] [++opt] [+cmd] {file} :split_f

Like :split but create a new window and start editing file {file} in it.
This behaves almost like a ":split" first, and then an ":edit" command, but the alternate file name in the original window is set to {file}.
If [+cmd] is given, execute the command when the file has been loaded +cmd.
Also see ++opt.
Make new window N high (default is to use half the existing height). Reduces the current window height to create room (and others, if the 'equalalways' option is set).

CTRL-W CTRL-V CTRL-W_CTRL-V

CTRL-W v CTRL-W_v

:[N]vs[plit] [++opt] [+cmd] [file] :vs :vsplit
Like :split, but split vertically. The windows will be
spread out horizontally if
1. a width was not specified,

2. 'equalalways' is set,

3. 'eadirection' isn't "ver", and

4. one of the other windows is wider than the current or new

window.
If N was given make the new window N columns wide, if
possible.
Note: In other places CTRL-Q does the same as CTRL-V, but here
it doesn't!

CTRL-W n CTRL-W_n

CTRL-W CTRL-N CTRL-W_CTRL-N

:[N]new [++opt] [+cmd] :new

Create a new window and start editing an empty file in it.
Make new window N high (default is to use half the existing
height). Reduces the current window height to create room
(and others, if the 'equalalways' option is set and
'eadirection' isn't "hor").
Also see ++opt and +cmd.
If 'fileformats' is not empty, the first format given will be
used for the new buffer. If 'fileformats' is empty, the
'fileformat' of the current buffer is used. This can be
overridden with the ++opt argument.
Autocommands are executed in this order:
1. WinLeave for the current window

2. WinEnter for the new window

3. BufLeave for the current buffer

4. BufEnter for the new buffer

This behaves like a ":split" first, and then an ":enew"
command.

:[N]new [++opt] [+cmd] {file}
Like :split_f, create a new window and start editing {file}.

:[N]vne[w] [++opt] [+cmd] [file] :vne :vnew
Like :new, but split vertically. If 'equalalways' is set
and 'eadirection' isn't "ver" the windows will be spread out
horizontally, unless a width was specified.

:[N]sv[iew] [++opt] [+cmd] [file] :sv :sview splitview
Same as ":split", but set 'readonly' option for this buffer.

:[N]sf[ind] [++opt] [+cmd] {file} :sf :sfi :sfind splitfind
Same as ":split", but search for {file} in 'path' like in
:find. Doesn't split if {file} is not found.

CTRL-W CTRL-^ CTRL-W_CTRL-^ CTRL-W_^
CTRL-W ^ Split the current window in two and edit the alternate file.
When a count N is given, split the current window and edit
buffer N. Similar to ":sp #" and ":sp #N", but it allows the
other buffer to be unnamed. This command matches the behavior
of CTRL-^, except that it splits a window first.

CTRL-W ge CTRL-W_ge

Detach the current window as an external window.
Only available when using a UI with ui-multigrid support.

Note that the 'splitbelow' and 'splitright' options influence where a new
window will appear.

E36

Creating a window will fail if there is not enough room. Every window needs
at least one screen line and column, sometimes more. Options 'winminheight'
and 'winminwidth' are relevant.

:vert :vertical
:vert[ical] {cmd}
Execute {cmd}. If it contains a command that splits a window,

it will be split vertically. For vertical wincmd = windows
will be equalized only vertically.
Doesn't work for :execute and :normal.

:hor :horizontal
:hor[izontal] {cmd}
Execute {cmd} in a horizontal split window. Supports these
commands:

- :wincmd =: equalize windows only horizontally.

- :terminal: open a terminal buffer in a split window.

- :checkhealth: open a healthcheck buffer in a split window.

:lefta[bove] {cmd} :lefta :leftabove
:abo[veleft] {cmd} :abo :aboveleft
Execute {cmd}. If it contains a command that splits a window,
it will be opened left (vertical split) or above (horizontal
split) the current window. Overrules 'splitbelow' and
'splitright'.
Doesn't work for :execute and :normal.

:rightb[elow] {cmd} :rightb :rightbelow
:bel[owright] {cmd} :bel :belowright
Execute {cmd}. If it contains a command that splits a window,
it will be opened right (vertical split) or below (horizontal
split) the current window. Overrules 'splitbelow' and
'splitright'.
Doesn't work for :execute and :normal.

:topleft E442
:to[pleft] {cmd}
Execute {cmd}. If it contains a command that splits a window,
it will appear at the top and occupy the full width of the Vim
window. When the split is vertical the window appears at the
far left and occupies the full height of the Vim window.
Doesn't work for :execute and :normal.

:bo :botright
:bo[tright] {cmd}
Execute {cmd}. If it contains a command that splits a window,
it will appear at the bottom and occupy the full width of the
Vim window. When the split is vertical the window appears at
the far right and occupies the full height of the Vim window.
Doesn't work for :execute and :normal.

These command modifiers can be combined to make a vertically split window
occupy the full height. Example:

```
:vertical topleft split tags
```

Opens a vertically split, full-height window on the "tags" file at the far
left of the Vim window.

Closing a window

:q[uit]
:{count}q[uit] :count_quit

CTRL-W q CTRL-W_q

CTRL-W CTRL-Q CTRL-W_CTRL-Q

Without {count}: Quit the current window. If {count} is
given quit the {count} window.

edit-window

When quitting the last edit window (not counting help or
preview windows), exit Vim.

When 'hidden' is set, and there is only one window for the
current buffer, it becomes hidden. When 'hidden' is not set,
and there is only one window for the current buffer, and the
buffer was changed, the command fails.
(**Note:** CTRL-Q does not work on all terminals).
If [count] is greater than the last window number the last
window will be closed:

```
:1quit  " quit the first window
:$quit  " quit the last window
:9quit  " quit the last window
        " if there are fewer than 9 windows opened
:-quit  " quit the previous window
:+quit  " quit the next window
:+2quit " quit the second next window
```

When closing a help window, and this is not the only window,
Vim will try to restore the previous window layout, see
:helpclose.

:q[uit]!
:{count}q[uit]!
Without {count}: Quit the current window. If {count} is given quit the {count} window
If this was the last window for a buffer, any changes to that buffer are lost. When quitting the last window (not counting help windows), exit Vim. The contents of the buffer are lost, even when 'hidden' is set.

:clo[se][!]
:{count}clo[se][!]
CTRL-W c CTRL-W_c :clo :close
Without {count}: Close the current window. If given close the {count} window.

When 'hidden' is set, or when the buffer was changed and the [!] is used, the buffer becomes hidden (unless there is another window editing it).

When there is only one edit-window in the current tabpage and there is another tabpage, this closes the current tabpage.

This command fails when: E444

- There is only one window on the screen.

- When 'hidden' is not set, [!] is not used, the buffer has

changes, and there is no other window on this buffer. Changes to the buffer are not written and won't get lost, so this is a "safe" command.

CTRL-W CTRL-C CTRL-W_CTRL-C

You might have expected that CTRL-W CTRL-C closes the current window, but that does not work, because the CTRL-C cancels the command.

:hide

:hid[e]
:{count}hid[e]
Without {count}: Quit the current window, unless it is the last window on the screen.
If {count} is given quit the {count} window.

The buffer becomes hidden (unless there is another window editing it or 'bufhidden' is unload, delete or wipe).
If the window is the last one in the current tabpage the tabpage is closed.

The value of 'hidden' is irrelevant for this command.
Changes to the buffer are not written and won't get lost, so this is a "safe" command.

:hid[e] {cmd} Execute {cmd} with 'hidden' set. The previous value of 'hidden' is restored after {cmd} has been executed.
Example:

```
:hide edit Makefile
```

This will edit "Makefile", and hide the current buffer if it has any changes.

:on[ly][!]
:{count}on[ly][!]
CTRL-W o CTRL-W_o E445
CTRL-W CTRL-O CTRL-W_CTRL-O :on :only
Make the current window the only one on the screen. All other windows are closed. For {count} see the :quit command above :count_quit.

When the 'hidden' option is set, all buffers in closed windows become hidden.

When 'hidden' is not set, and the 'autowrite' option is set, modified buffers are written. Otherwise, windows that have buffers that are modified are not removed, unless the [!] is given, then they become hidden. But modified buffers are never abandoned, so changes cannot get lost.

:fc :fclose

:[count]fc[lose][!]
Close [count] floating windows with the highest zindex values.
'!' to close all floating windows.

4. Moving cursor to other windows window-move-cursor

CTRL-W <Down> CTRL-W <Down>

CTRL-W CTRL-J CTRL-W_CTRL-J CTRL-W_j
CTRL-W j Move cursor to Nth window below current one. Uses the cursor position to select between alternatives.

CTRL-W <Up> CTRL-W_<Up>

CTRL-W CTRL-K CTRL-W_CTRL-K CTRL-W_k
CTRL-W k Move cursor to Nth window above current one. Uses the cursor position to select between alternatives.

CTRL-W <Left> CTRL-W_<Left>

CTRL-W CTRL-H CTRL-W_CTRL-H

CTRL-W <BS> CTRL-W_<BS> CTRL-W_h
CTRL-W h Move cursor to Nth window left of current one. Uses the cursor position to select between alternatives.

CTRL-W <Right> CTRL-W_<Right>

CTRL-W CTRL-L CTRL-W_CTRL-L CTRL-W_l
CTRL-W l Move cursor to Nth window right of current one. Uses the cursor position to select between alternatives.

CTRL-W w CTRL-W_w CTRL-W_CTRL-W
CTRL-W CTRL-W Without count: move cursor to the focusable window below/right of the current one. If none, go to the top-left window. With count: go to Nth window (numbered top-left to bottom-right), skipping unfocusable windows. To obtain the window number see bufwinnr() and winnr(). When N is larger than the number of windows go to the last focusable window.

CTRL-W_W

CTRL-W W Without count: move cursor to the focusable window above/left of current one. If none, go to the bottom-right window. With count: go to Nth window, like CTRL-W w.

CTRL-W t CTRL-W_t CTRL-W_CTRL-T
CTRL-W CTRL-T Move cursor to top-left window.

CTRL-W b CTRL-W_b CTRL-W_CTRL-B
CTRL-W CTRL-B Move cursor to bottom-right window.

CTRL-W p CTRL-W_p CTRL-W_CTRL-P
CTRL-W CTRL-P Go to previous (last accessed) window.

CTRL-W_P E441
CTRL-W P Go to preview window. When there is no preview window this is an error.

If Visual mode is active and the new window is not for the same buffer, the Visual mode is ended. If the window is on the same buffer, the cursor position is set to keep the same Visual area selected.

:winc :wincmd
These commands can also be executed with ":wincmd":

:[count]winc[md] {arg}
:winc[md] [count] {arg}
  Like executing CTRL-W [count] {arg}. Example:

```
:wincmd j
```

  Moves to the window below the current one.
  This command is useful when a Normal mode cannot be used (for the CursorHold autocommand event). Or when a Normal mode command is inconvenient.
  The count can also be a window number. Example:

```
:exe nr .. "wincmd w"
```

  This goes to window "nr".

Note: All CTRL-W commands can also be executed with :wincmd, for those places where a Normal mode command can't be used or is inconvenient (e.g. in a browser-based terminal).

5. Moving windows around window-moving

CTRL-W r CTRL-W_r CTRL-W_CTRL-R E443
CTRL-W CTRL-R Rotate windows downwards/rightwards. The first window becomes the second one, the second one becomes the third one, etc. The last window becomes the first window. The cursor remains in the same window.
  This only works within the row or column of windows that the current window is in.

CTRL-W_R

CTRL-W R Rotate windows upwards/leftwards. The second window becomes the first one, the third one becomes the second one, etc. The first window becomes the last window. The cursor remains in the same window.
  This only works within the row or column of windows that the current window is in.

CTRL-W x CTRL-W_x CTRL-W_CTRL-X
CTRL-W CTRL-X Without count: Exchange current window with next one. If there is no next window, exchange with previous window.

With count: Exchange current window with Nth window (first
window is 1). The cursor is put in the other window.
When vertical and horizontal window splits are mixed, the
exchange is only done in the row or column of windows that the
current window is in.

The following commands can be used to change the window layout. For example,
when there are two vertically split windows, CTRL-W K will change that in
horizontally split windows. CTRL-W H does it the other way around.

CTRL-W_K

CTRL-W K Move the current window to be at the very top, using the full
width of the screen. This works like :topleft split, except
it is applied to the current window and no new window is
created.

CTRL-W_J

CTRL-W J Move the current window to be at the very bottom, using the
full width of the screen. This works like :botright split,
except it is applied to the current window and no new window
is created.

CTRL-W_H

CTRL-W H Move the current window to be at the far left, using the
full height of the screen. This works like
:vert topleft split, except it is applied to the current
window and no new window is created.

CTRL-W_L

CTRL-W L Move the current window to be at the far right, using the full
height of the screen. This works like :vert botright split,
except it is applied to the current window and no new window
is created.

CTRL-W_T

CTRL-W T Move the current window to a new tabpage. This fails if
there is only one window in the current tabpage.
This works like :tab split, except the previous window is
closed.
When a count is specified the new tabpage will be opened
before the tabpage with this index. Otherwise it comes after
the current tabpage.

6. Window resizing window-resize

CTRL-W_=

CTRL-W = Make all windows (almost) equally high and wide, but use
'winheight' and 'winwidth' for the current window. Windows
with 'winfixheight' set keep their height, and windows with
'winfixwidth' set keep their width.

To equalize only vertically (each window equally high) use:

```
:vertical wincmd =
```

To equalize only horizontally (each window equally wide) use:

```
:horizontal wincmd =
```

To equalize windows when Nvim itself is resized, see
VimResized.

:res[ize] -N :res :resize CTRL-W_-
CTRL-W - Decrease current window height by N (default 1).
If used after :vertical: decrease width by N.

:res[ize] +N CTRL-W_+

CTRL-W + Increase current window height by N (default 1).
If used after :vertical: increase width by N.

:res[ize] [N]
CTRL-W CTRL-_ CTRL-W_CTRL-_ CTRL-W__
CTRL-W _ Set current window height to N (default: highest possible).

:{winnr}res[ize] [+-]N
Like :resize above, but apply the size to window {winnr}
instead of the current window. See also nvim_win_resize().

z{nr}<CR> Set current window height to {nr}.

CTRL-W_<

CTRL-W < Decrease current window width by N (default 1).

CTRL-W_>

CTRL-W > Increase current window width by N (default 1).

:vert[ical] res[ize] [N] :vertical-resize CTRL-W_bar
CTRL-W | Set current window width to N (default: widest possible).

You can also resize a window by dragging a status line up or down with the mouse. Or by dragging a vertical separator line left or right. This only works if the version of Vim that is being used supports the mouse and the 'mouse' option has been set to enable it.

The option 'winheight' ('wh') is used to set the minimal window height of the current window. This option is used each time another window becomes the current window. If the option is '0', it is disabled. Set 'winheight' to a very large value, e.g., '9999', to make the current window always fill all available space. Set it to a reasonable value, e.g., '10', to make editing in the current window comfortable.

The equivalent 'winwidth' ('wiw') option is used to set the minimal width of the current window.

When the option 'equalalways' ('ea') is set, all the windows are automatically made the same size after splitting or closing a window. If you don't set this option, splitting a window will reduce the size of the current window and leave the other windows the same. When closing a window, the extra lines are given to the window above it.

The 'eadirection' option limits the direction in which the 'equalalways' option is applied. The default "both" resizes in both directions. When the value is "ver" only the heights of windows are equalized. Use this when you have manually resized a vertically split window and want to keep this width. Likewise, "hor" causes only the widths of windows to be equalized.

The option 'cmdheight' ('ch') is used to set the height of the command-line. If you are annoyed by the hit-enter prompt for long messages, set this option to 2 or 3.

If there is only one window, resizing that window will also change the command line height. If there are several windows, resizing the current window will also change the height of the window below it (and sometimes the window above it).

The minimal height and width of a window is set with 'winminheight' and 'winminwidth'. These are hard values, a window will never become smaller.

WinScrolled and WinResized autocommands

win-scrolled-resized

If you want to get notified of changes in window sizes, the WinResized autocommand event can be used.
If you want to get notified of text in windows scrolling vertically or horizontally, the WinScrolled autocommand event can be used. This will also trigger in window size changes.
Exception: the events will not be triggered when the text scrolls for 'incsearch'.

WinResized-event

The WinResized event is triggered after updating the display, several windows may have changed size then. A list of the IDs of windows that changed since last time is provided in the v:event.windows variable, for example:

```
        [1003, 1006]
```

WinScrolled-event

The WinScrolled event is triggered after WinResized, and also if a window was scrolled. That can be vertically (the text at the top of the window changed) or horizontally (when 'wrap' is off or when the first displayed part of the first line changes). Note that WinScrolled will trigger many more times than WinResized, it may slow down editing a bit.

The information provided by WinScrolled is a dictionary for each window that has changes, using the window ID as the key, and a total count of the changes with the key "all". Example value for v:event:

```
{
   all:  {width: 0, height:  2, leftcol: 0, skipcol: 0, topline: 1, topfill: 0},
   1003: {width: 0, height: -1, leftcol: 0, skipcol: 0, topline: 0, topfill: 0},
   1006: {width: 0, height:  1, leftcol: 0, skipcol: 0, topline: 1, topfill: 0},
}
```

Note that the "all" entry has the absolute values of the individual windows accumulated.

If you need more information about what changed, or you want to "debounce" the events (not handle every event to avoid doing too much work), you may want to use the winlayout() and getwininfo() functions.

WinScrolled and WinResized do not trigger when the first autocommand is added, only after the first scroll or resize. They may trigger when switching to another tabpage.

The commands executed are expected to not cause window size or scroll changes. If this happens anyway, the event will trigger again very soon. In other words: Just before triggering the event, the current sizes and scroll positions are stored and used to decide whether there was a change.

7. Argument and buffer list commands

buffer-list

args list	buffer list	meaning

```
1. :[N]argument [N]     11. :[N]buffer [N]      to arg/buf N
2. :[N]next [file ..]   12. :[N]bnext [N]       to Nth next arg/buf
3. :[N]Next [N] 13. :[N]bNext [N]       to Nth previous arg/buf
4. :[N]previous [N]     14. :[N]bprevious [N]   to Nth previous arg/buf
5. :rewind / :first     15. :brewind / :bfirst  to first arg/buf
6. :last                16. :blast              to last arg/buf
7. :all         17. :ball               edit all args/buffers
                        18. :unhide             edit all loaded buffers
                        19. :[N]bmod [N]        to Nth modified buf
```

split & args list	split & buffer list	meaning

```
21. :[N]sargument [N]   31. :[N]sbuffer [N]     split + to arg/buf N
22. :[N]snext [file ..] 32. :[N]sbnext [N]      split + to Nth next arg/buf
23. :[N]sNext [N]       33. :[N]sbNext [N]      split + to Nth previous arg/buf
24. :[N]sprevious [N]   34. :[N]sbprevious [N]  split + to Nth previous arg/buf
25. :srewind / :sfirst   35. :sbrewind / :sbfirst split + to first arg/buf
26. :slast               36. :sblast            split + to last arg/buf
27. :sall                37. :sball             edit all args/buffers
                         38. :sunhide           edit all loaded buffers
                         39. :[N]sbmod [N]      split + to Nth modified buf

40. :args               list of arguments
41. :buffers            list of buffers
```

The meaning of [N] depends on the command:
 [N] is the number of buffers to go forward/backward on 2/12/22/32,
     3/13/23/33, and 4/14/24/34
 [N] is an argument number, defaulting to current argument, for 1 and 21
 [N] is a buffer number, defaulting to current buffer, for 11 and 31
 [N] is a count for 19 and 39

Note: ":next" is an exception, because it must accept a list of file names
for compatibility with Vi.

The argument list and multiple windows

The current position in the argument list can be different for each window.
Remember that when doing ":e file", the position in the argument list stays
the same, but you are not editing the file at that position. To indicate
this, the file message (and the title, if you have one) shows
"(file (N) of M)", where "(N)" is the current position in the file list, and
"M" the number of files in the file list.

All the entries in the argument list are added to the buffer list. Thus, you
can also get to them with the buffer list commands, like ":bnext".

:[N]al[l][!] [N] :al :all :sal :sall
:[N]sal[l][!] [N]
Rearrange the screen to open one window for each argument.
All other windows are closed. When a count is given, this is
the maximum number of windows to open.
With the :tab modifier open a tabpage for each argument.
When there are more arguments than 'tabpagemax' further ones
become split windows in the last tabpage.
When the 'hidden' option is set, all buffers in closed windows
become hidden.
When 'hidden' is not set, and the 'autowrite' option is set,
modified buffers are written. Otherwise, windows that have
buffers that are modified are not removed, unless the [!] is
given, then they become hidden. But modified buffers are
never abandoned, so changes cannot get lost.
[N] is the maximum number of windows to open. 'winheight'
also limits the number of windows opened ('winwidth' if
:vertical was prepended).
Buf/Win Enter/Leave autocommands are not executed for the new
windows here, that's only done when they are really entered.
If autocommands change the window layout while this command is
busy an error will be given. E249

:[N]sa[rgument][!] [++opt] [+cmd] [N] :sa :sargument
Short for ":split | argument [N]": split window and go to Nth
argument. But when there is no such argument, the window is
not split. Also see ++opt and +cmd.

:[N]sn[ext][!] [++opt] [+cmd] [file ..] :sn :snext
Short for ":split | [N]next": split window and go to Nth next
argument. But when there is no next file, the window is not
split. Also see ++opt and +cmd.

:[N]spr[evious][!] [++opt] [+cmd] [N] :spr :sprevious
:[N]sN[ext][!] [++opt] [+cmd] [N] :sN :sNext
Short for ":split | [N]Next": split window and go to Nth
previous argument. But when there is no previous file, the
window is not split. Also see ++opt and +cmd.

:sre :srewind
:sre[wind][!] [++opt] [+cmd]
Short for ":split | rewind": split window and go to first argument. But when there is no argument list, the window is not split. Also see ++opt and +cmd.

:sfir :sfirst
:sfir[st] [++opt] [+cmd]
Same as ":srewind".

:sla :slast
:sla[st][!] [++opt] [+cmd]
Short for ":split | last": split window and go to last argument. But when there is no argument list, the window is not split. Also see ++opt and +cmd.

:dr :drop
:dr[op] [++opt] [+cmd] {file} ..
Edit the first {file} in a window.

- If the file is already open in a window change to that window.
- If the file is not open in a window edit the file in the current window. If the current buffer can't be abandoned, the window is split first.
- Windows that are not in the argument list or are not full width will be closed if possible.

The argument-list is set, like with the :next command.
The purpose of this command is that it can be used from a program that wants Vim to edit another file, e.g., a debugger.
When using the :tab modifier each argument is opened in a tabpage. The last window is used if it's empty.
Also see ++opt and +cmd.

8. Do a command in all buffers or windows list-repeat

:windo

:[range]windo {cmd} Execute {cmd} in each focusable window, or only for windows in a given [range] of window numbers. It works like doing this:

```
CTRL-W t
:{cmd}
CTRL-W w
:{cmd}
etc.
```

This only operates in the current tabpage.
When an error is detected on one window, further windows will not be visited.
The last window (or where an error occurred) becomes the current window.
{cmd} can contain '|' to concatenate several commands.
{cmd} must not open or close windows or reorder them.

Also see :tabdo, :argdo, :bufdo, :cdo, :ldo, :cfdo and :lfdo.

:bufdo

:[range]bufdo[!] {cmd} Execute {cmd} in each buffer in the buffer list or if [range] is given only for buffers for which their buffer number is in the [range]. It works like doing this:

```
:bfirst
:{cmd}
:bnext
:{cmd}
etc.
```

When the current file can't be abandoned and the [!] is not present, the command fails.
When an error is detected on one buffer, further buffers will not be visited.
Unlisted buffers are skipped.
The last buffer (or where an error occurred) becomes the current buffer.
{cmd} can contain '|' to concatenate several commands.
{cmd} must not delete buffers or add buffers to the buffer list.
Note: While this command is executing, the Syntax autocommand event is disabled by adding it to 'eventignore'. This considerably speeds up editing each buffer.

Also see :tabdo, :argdo, :windo, :cdo, :ldo,
:cfdo and :lfdo.

Examples:

```
:windo set nolist foldcolumn=0 | normal! zn
```

This resets the 'list' option and disables folding in all windows.

```
:bufdo set fileencoding= | update
```

This resets the 'fileencoding' in each buffer and writes it if this changed
the buffer. The result is that all buffers will use the 'encoding' encoding
(if conversion succeeds).

9. Tag or file name under the cursor

window-tag

:sta :stag

:sta[g][!] [tagname]
Does ":tag[!] [tagname]" and splits the window for the found
tag. Refer to 'switchbuf' to jump to a tag in a vertically
split window or a new tabpage. See also :tag.

CTRL-W] CTRL-W_] CTRL-W_CTRL-]
CTRL-W CTRL-] Split current window in two. Use identifier under cursor as a
tag and jump to it in the new upper window.
In Visual mode uses the Visually selected text as a tag.
Make new window N high. Refer to 'switchbuf' to jump to a tag
in a vertically split window or a new tabpage.

CTRL-W g]

CTRL-W g] Split current window in two. Use identifier under cursor as a
tag and perform ":tselect" on it in the new upper window.
In Visual mode uses the Visually selected text as a tag.
Make new window N high.

CTRL-W g CTRL-]

CTRL-W g CTRL-] Split current window in two. Use identifier under cursor as a
tag and perform ":tjump" on it in the new upper window.
In Visual mode uses the Visually selected text as a tag.
Make new window N high.

CTRL-W f CTRL-W_f CTRL-W_CTRL-F
CTRL-W CTRL-F Split current window in two. Edit file name under cursor.
Like ":split gf", but window isn't split if the file does not
exist.
Uses the 'path' variable as a list of directory names where to
look for the file. Also the path for current file is
used to search for the file name.
If the name is a hypertext link that looks like
"type://machine/path", only "/path" is used.
If a count is given, the count'th matching file is edited.

CTRL-W F CTRL-W_F

Split current window in two. Edit file name under cursor and
jump to the line number following the file name. See gF for
details on how the line number is obtained.

CTRL-W gf CTRL-W_gf

Open a new tabpage and edit the file name under the cursor.
Like "tab split" and "gf", but the new tabpage isn't created
if the file does not exist.

CTRL-W gF CTRL-W_gF

Open a new tabpage and edit the file name under the cursor
and jump to the line number following the file name. Like
"tab split" and "gF", but the new tabpage isn't created if
the file does not exist.

CTRL-W gt CTRL-W_gt

Go to next tabpage, same as gt.

CTRL-W gT CTRL-W_gT

Go to previous tabpage, same as gT.

Also see CTRL-W_CTRL-I: open window for an included file that includes
the keyword under the cursor.

10. The preview window

preview-window

The preview window is a special window to show (preview) another file. It is
normally a small window used to show an include file or definition of a
function.

There can be only one preview window (per tabpage). It is created with one
of the commands below. The 'previewheight' option can be set to specify the
height of the preview window when it's opened. The 'previewwindow' option is

set in the preview window to be able to recognize it. The 'winfixheight' option is set to have it keep the same height when opening/closing other windows.

preview-popup

Alternatively, a floating window can be used by setting the 'previewpopup' option. When set, it overrules the 'previewwindow' and 'previewheight' settings.

:pt :ptag

:pt[ag][!] [tagname]
Does ":tag[!] [tagname]" and shows the found tag in a "Preview" window without changing the current buffer or cursor position. If a "Preview" window already exists, it is re-used (like a help window is). If a new one is opened, 'previewheight' is used for the height of the window. See also :tag.
See below for an example. CursorHold-example
Small difference from :tag: When [tagname] is equal to the already displayed tag, the position in the matching tag list is not reset. This makes the CursorHold example work after a :ptnext.

CTRL-W z CTRL-W_z

CTRL-W CTRL-Z CTRL-W_CTRL-Z :pc :pclose

:pc[lose][!]
Close any "Preview" window currently open. When the 'hidden' option is set, or when the buffer was changed and the [!] is used, the buffer becomes hidden (unless there is another window editing it). The command fails if any "Preview" buffer cannot be closed. See also :close.

:pp :ppop

:[count]pp[op][!]
Does ":[count]pop[!]" in the preview window. See :pop and :ptag.

CTRL-W } CTRL-W_}

Use identifier under cursor as a tag and perform a :ptag on it. Make the new Preview window (if required) N high. If N is not given, 'previewheight' is used.

CTRL-W g } CTRL-W_g}

Use identifier under cursor as a tag and perform a :ptjump on it. Make the new Preview window (if required) N high. If N is not given, 'previewheight' is used.

:pb :pbuffer

:[N]pb[uffer][!] [+cmd] [N]
Edit buffer [N] from the buffer list in the preview window. If [N] is not given, the current buffer remains being edited. See :buffer-! for [!]. This will also edit a buffer that is not in the buffer list, without setting the 'buflisted' flag. Also see +cmd.

:ped :pedit

:ped[it][!] [++opt] [+cmd] {file}
Edit {file} in the preview window. The preview window is opened like with :ptag. The current window and cursor position isn't changed. Useful example:

```
:pedit +/fputc /usr/include/stdio.h
```

Also see ++opt and +cmd.

:ps :psearch

:[range]ps[earch][!] [count] [/]pattern[/]
Works like :ijump but shows the found match in the preview window. The preview window is opened like with :ptag. The current window and cursor position isn't changed. Useful example:

```
:psearch popen
```

Like with the :ptag command, you can use this to automatically show information about the word under the cursor. This is less clever than using :ptag, but you don't need a tags file and it will also find matches in system include files. Example:

```
:au! CursorHold *.[ch] ++nested exe "silent! psearch " .. expand("<cword>")
```

Warning: This can be slow.

Example CursorHold-example

```
:au! CursorHold *.[ch] ++nested exe "silent! ptag " .. expand("<cword>")
```

This will cause a ":ptag" to be executed for the keyword under the cursor, when the cursor hasn't moved for the time set with 'updatetime'. "++nested" makes other autocommands be executed, so that syntax highlighting works in the

preview window. The "silent!" avoids an error message when the tag could not
be found. Also see CursorHold. To disable this again:

```
:au! CursorHold
```

A nice addition is to highlight the found tag, avoid the ":ptag" when there
is no word under the cursor, and a few other things:

```
:au! CursorHold *.[ch] ++nested call PreviewWord()
:func PreviewWord()
:  if &previewwindow                    " don't do this in the preview window
:    return
:  endif
:  let w = expand("<cword>")             " get the word under cursor
:  if w =~ '\a'                  " if the word contains a letter
:
:    " Delete any existing highlight before showing another tag
:    silent! wincmd P                    " jump to preview window
:    if &previewwindow           " if we really get there...
:      match none                        " delete existing highlight
:      wincmd p                  " back to old window
:    endif
:
:    " Try displaying a matching tag for the word under the cursor
:    try
:       exe "ptag " .. w
:    catch
:      return
:    endtry
:
:    silent! wincmd P                    " jump to preview window
:    if &previewwindow           " if we really get there...
:        if has("folding")
:          silent! .foldopen             " don't want a closed fold
:        endif
:        call search("$", "b")           " to end of previous line
:        let w = substitute(w, '\\', '\\\\', "")
:        call search('\<\V' .. w .. '\>')       " position cursor on match
:        " Add a match highlight to the word at this position
:      hi previewWord term=bold ctermbg=green guibg=green
:        exe 'match previewWord "\%' .. line(".") .. 'l\%' .. col(".") .. 'c\k*"'
:      wincmd p                  " back to old window
:    endif
:  endif
:endfun
```

11. Using hidden buffers

buffer-hidden

A hidden buffer is not displayed in a window, but is still loaded into memory.
This makes it possible to jump from file to file, without the need to read or
write the file every time you get another buffer in a window.

:buffer-!

If the option 'hidden' ('hid') is set, abandoned buffers are kept for all
commands that start editing another file: ":edit", ":next", ":tag", etc. The
commands that move through the buffer list sometimes make the current buffer
hidden although the 'hidden' option is not set. This happens when a buffer is
modified, but is forced (with '!') to be removed from a window, and
'autowrite' is off or the buffer can't be written.

You can make a hidden buffer not hidden by starting to edit it with any
command, or by deleting it with the ":bdelete" command.

The 'hidden' is global, it is used for all buffers. The 'bufhidden' option
can be used to make an exception for a specific buffer. It can take these
values:

<empty>	Use the value of 'hidden'.
hide	Hide this buffer, also when 'hidden' is not set.
unload	Don't hide but unload this buffer, also when 'hidden' is set.
delete	Delete the buffer.

hidden-quit

When you try to quit Vim while there is a hidden, modified buffer, you will
get an error message and Vim will make that buffer the current buffer. You
can then decide to write this buffer (":wq") or quit without writing (":q!").
Be careful: there may be more hidden, modified buffers!

A buffer can also be unlisted. This means it exists, but it is not in the
list of buffers. unlisted-buffer

:files[!] [flags] :files

:buffers[!] [flags] :buffers :ls
:ls[!] [flags]
Show all buffers. Example:

1 #h "/test/text" line 1

2u "asdf" line 0
3 %a + "version.c" line 1

When the [!] is included the list will show unlisted buffers
(the term "unlisted" is a bit confusing then...).

Each buffer has a unique number. That number will not change,
thus you can always go to a specific buffer with ":buffer N"
or "N CTRL-^", where N is the buffer number.

For the file name these special values are used:
"[Prompt]" prompt-buffer
"[Scratch]" 'buftype' is "nofile"
"[No Name]" no file name specified

Indicators (chars in the same column are mutually exclusive):
u unlisted-buffer (only displayed when [!] is used)
% buffer in the current window
alternate-file buffer for :e# and CTRL-^
a an active buffer: loaded and visible
h hidden-buffer: loaded, but not displayed in a window
\- 'nomodifiable' buffer
= readonly buffer
R terminal buffer with a Running job (open channel)
F terminal buffer with a Finished job (closed channel)
? terminal buffer without a job: :terminal NONE
\+ 'modified' buffer
x buffer with read errors

[flags] can be a combination of the following characters,
which restrict the buffers to be listed:
\+ modified buffers
\- 'nomodifiable' buffers
= readonly buffers
a active buffers
u unlisted buffers (overrides the "!")
h hidden buffers
x buffers with a read error
% current buffer
alternate buffer
R terminal buffers with a Running job
F terminal buffers with a Finished job
t show time last used and sort buffers
Combining flags means they are "and"ed together, e.g.:
h+ hidden buffers which are modified
a+ active buffers which are modified

When using :filter the pattern is matched against the
displayed buffer name, e.g.:

```
filter /\.vim/ ls
```

:bad :badd

:bad[d] [+lnum] {fname}
Add file name {fname} to the buffer list, without loading it,
if it wasn't listed yet. If the buffer was previously
deleted, not wiped, it will be made listed again.
If "lnum" is specified, the cursor will be positioned at that
line when the buffer is first entered. Note that other
commands after the + will be ignored.

:balt

:balt [+lnum] {fname}
Like :badd and also set the alternate file for the current
window to {fname}.

:[N]bd[elete][!] :bd :bdel :bdelete E516
:bd[elete][!] [N]
Unload buffer [N] (default: current buffer) and delete it from
the buffer list. If the buffer was changed, this fails,
unless [!] is specified, in which case changes are lost.
The file remains unaffected. Any windows for this buffer are
closed. If buffer [N] is the current buffer, another buffer
will be displayed instead. This is the most recent entry in
the jump list that points into a loaded buffer.

Actually, the buffer isn't completely deleted, it is removed
from the buffer list unlisted-buffer and option values,
variables and mappings/abbreviations for the buffer are
cleared. If [N] is the last listed buffer in a window (i.e.,
there is no other listed buffer to switch to), the buffer is
emptied instead of being unloaded. The window is not closed,
and the buffer may be reused as a new buffer buffer-reuse.
This ensures every window always has a valid buffer.

Examples:

```
:.,$-bdelete " delete buffers from the current one to
" last but one
:%bdelete " delete all buffers
```

:bd[elete][!] {bufname} E93 E94
Like ":bdelete[!] [N]", but buffer given by name, see {bufname}.

:bd[elete][!] N1 N2 ...
Do ":bdelete[!]" for buffer N1, N2, etc. The arguments can be buffer numbers or buffer names (but not buffer names that are a number). Insert a backslash before a space in a buffer name.

:N,Mbd[elete][!]
Do ":bdelete[!]" for all buffers in the range N to M inclusive.

:[N]bw[ipeout][!] :bw :bwipe :bwipeout E517
:bw[ipeout][!] {bufname}
:N,Mbw[ipeout][!]
:bw[ipeout][!] N1 N2 ...
Like :bdelete, but really delete the buffer. Everything related to the buffer is lost. All marks in this buffer become invalid, option settings are lost, the jumplist and tagstack data will be purged, etc. Don't use this unless you know what you are doing. Examples:

```
:.+,$bwipeout   " wipe out all buffers after the current
                " one
:%bwipeout          " wipe out all buffers
```

:[N]bun[load][!] :bun :bunload E515
:bun[load][!] [N]
Unload buffer [N] (default: current buffer). The memory allocated for this buffer will be freed. The buffer remains in the buffer list.
If the buffer was changed, this fails, unless when [!] is specified, in which case the changes are lost.
Any windows for this buffer are closed. If buffer [N] is the current buffer, another buffer will be displayed instead.
This is the most recent entry in the jump list that points into a loaded buffer.

:bun[load][!] {bufname}
Like ":bunload[!] [N]", but buffer given by name.
Also see {bufname}.

:N,Mbun[load][!]
Do ":bunload[!]" for all buffers in the range N to M inclusive.

:bun[load][!] N1 N2 ...
Do ":bunload[!]" for buffer N1, N2, etc. The arguments can be buffer numbers or buffer names (but not buffer names that are a number). Insert a backslash before a space in a buffer name.

:[N]b[uffer][!] [+cmd] [N] :b :bu :buf :buffer E86
Edit buffer [N] from the buffer list. If [N] is not given, the current buffer remains being edited. See :buffer-! for [!]. This will also edit a buffer that is not in the buffer list, without setting the 'buflisted' flag.
Also see +cmd.

:[N]b[uffer][!] [+cmd] {bufname} {bufname}

Edit buffer for {bufname} from the buffer list. A partial name also works, so long as it is unique in the list of buffers.
Note that a buffer whose name is a number cannot be referenced by that name; use the buffer number instead. Same is true if the buffer name starts with a +, it will be interpreted as the start of a +cmd.
Insert a backslash before a space in a buffer name.
See :buffer-! for [!].
This will also edit a buffer that is not in the buffer list, without setting the 'buflisted' flag.
Also see +cmd.

:[N]sb[uffer] [+cmd] [N] :sb :sbuffer
Split window and edit buffer [N] from the buffer list. If [N] is not given, the current buffer is edited. Respects the "useopen" setting of 'switchbuf' when splitting. This will also edit a buffer that is not in the buffer list, without setting the 'buflisted' flag.
Also see +cmd.

:[N]sb[uffer] [+cmd] {bufname}
Split window and edit buffer for {bufname} from the buffer list. This will also edit a buffer that is not in the buffer list, without setting the 'buflisted' flag.
Note: If what you want to do is split the buffer, make a copy under another name, you can do it this way:

```
:w foobar | sp #
```

Also see +cmd.

:[N]bn[ext][!] [+cmd] [N] :bn :bnext E87
Go to [N]th next buffer in buffer list. [N] defaults to one.
Wraps around the end of the buffer list.
See :buffer-! for [!].
Also see +cmd.
If you are in a help buffer, this takes you to the next help
buffer (if there is one). Similarly, if you are in a normal
(non-help) buffer, this takes you to the next normal buffer.
This is so that if you have invoked help, it doesn't get in
the way when you're browsing code/text buffers. The next
three commands also work like this.

]b

]b Mapped to :bnext. default-mappings

:sbn :sbnext
:[N]sbn[ext] [+cmd] [N]
Split window and go to [N]th next buffer in buffer list.
Wraps around the end of the buffer list. Uses 'switchbuf'
Also see +cmd.

:[N]bN[ext][!] [+cmd] [N] :bN :bNext :bp :bprevious E88

[b

[b Mapped to :bprevious. default-mappings

:[N]bp[revious][!] [+cmd] [N]
Go to [N]th previous buffer in buffer list. [N] defaults to
one. Wraps around the start of the buffer list.
See :buffer-! for [!] and 'switchbuf'.
Also see +cmd.

:[N]sbN[ext] [+cmd] [N] :sbN :sbNext :sbp :sbprevious
:[N]sbp[revious] [+cmd] [N]
Split window and go to [N]th previous buffer in buffer list.
Wraps around the start of the buffer list.
Uses 'switchbuf'.
Also see +cmd.

:br[ewind][!] [+cmd] :br :bre :brewind
Go to first buffer in buffer list. If the buffer list is
empty, go to the first unlisted buffer.
See :buffer-! for [!].

[B

[B Mapped to :brewind. default-mappings

:bf[irst] [+cmd] :bf :bfirst
Same as :brewind.
Also see +cmd.

:sbr[ewind] [+cmd] :sbr :sbrewind
Split window and go to first buffer in buffer list. If the
buffer list is empty, go to the first unlisted buffer.
Respects the 'switchbuf' option.
Also see +cmd.

:sbf[irst] [+cmd] :sbf :sbfirst
Same as ":sbrewind".

:bl[ast][!] [+cmd] :bl :blast
Go to last buffer in buffer list. If the buffer list is
empty, go to the last unlisted buffer.
See :buffer-! for [!].

]B

]B Mapped to :blast. default-mappings

:sbl[ast] [+cmd] :sbl :sblast
Split window and go to last buffer in buffer list. If the
buffer list is empty, go to the last unlisted buffer.
Respects 'switchbuf' option.

:[N]bm[odified][!] [+cmd] [N] :bm :bmodified E84
Go to [N]th next modified buffer. **Note:** this command also
finds unlisted buffers. If there is no modified buffer the
command fails.

:[N]sbm[odified] [+cmd] [N] :sbm :sbmodified
Split window and go to [N]th next modified buffer.
Respects 'switchbuf' option.
Note: this command also finds buffers not in the buffer list.

:[N]unh[ide] [N] :unh :unhide :sun :sunhide
:[N]sun[hide] [N]
Rearrange the screen to open one window for each loaded buffer
in the buffer list. When a count is given, this is the
maximum number of windows to open.

:[N]ba[ll] [N] :ba :ball :sba :sball
:[N]sba[ll] [N] Rearrange the screen to open one window for each buffer in
the buffer list. When a count is given, this is the maximum
number of windows to open. 'winheight' also limits the number
of windows opened ('winwidth' if :vertical was prepended).
Buf/Win Enter/Leave autocommands are not executed for the new
windows here, that's only done when they are really entered.

When the :tab modifier is used new windows are opened in a
new tab, up to 'tabpagemax'.

Note: All the commands above that start editing another buffer, keep the 'readonly' flag as it was. This differs from the ":edit" command, which sets the 'readonly' flag each time the file is read.

12. Special kinds of buffers special-buffers

Instead of containing the text of a file, buffers can also be used for other purposes. A few options can be set to change the behavior of a buffer:

'bufhidden'	what happens when the buffer is no longer displayed in a window.
'buftype'	what kind of a buffer this is
'swapfile'	whether the buffer will have a swap file
'buflisted'	buffer shows up in the buffer list

A few useful kinds of a buffer:

quickfix — Used to contain the error list or the location list. See :cwindow and :lwindow. This command sets the 'buftype' option to "quickfix". You are not supposed to change this! 'swapfile' is off.

help — Contains a help file. Will only be created with the :help command. The flag that indicates a help buffer is internal and can't be changed. The 'buflisted' option will be reset for a help buffer.

terminal — A terminal window buffer, see terminal. The contents cannot be read or changed until the job ends.

directory — Displays directory contents. Can be used by a file explorer plugin. The buffer is created with these settings:

```
:setlocal buftype=nowrite
:setlocal bufhidden=
:setlocal noswapfile
```

The buffer name is the name of the directory and is adjusted when using the :cd command.

scratch-buffer

scratch — Contains text that can be discarded at any time. It is kept when closing the window, it must be deleted explicitly. Settings:

```
:setlocal buftype=nofile
:setlocal bufhidden=hide
:setlocal noswapfile
```

The buffer name can be used to identify the buffer, if you give it a meaningful name.

unlisted-buffer

unlisted — The buffer is not in the buffer list. It is not used for normal editing, but to show a help file, remember a file name or marks. The ":bdelete" command will also set this option, thus it doesn't completely delete the buffer. Settings:

```
:setlocal nobuflisted
```

Builtin plugin: zip

Nvim opens a read-only listing when :edit is used with a zip archive. The listing is a dir buffer with 'filetype' set to "zip"; entries open as read-only zip://{archive}/{path} buffers. Requires the unzip executable.

Recognized extensions include zip, jar, apk, epub, Office and OpenDocument formats, whl, xpi, pkpass, and cbz. Remote archives are fetched by vim.net.

Reading an entry from an encrypted archive prompts for its password.

g:loaded_nvim_zip_plugin

To disable the built-in zip browser, set this before startup:

```
vim.g.loaded_nvim_zip_plugin = 1
```

MAPPINGS zip-mappings

The standard dir-buffer-mappings, plus:

- x extracts the entry under the cursor into the current directory, discarding its path inside the archive. It never overwrites an existing file.

At the archive root, - opens the containing filesystem directory.

LIMITATIONS zip-limitations

Archive entries cannot be updated. Remote archives are not refetched on reload, and there is no PowerShell fallback, no custom backend command, and no support for the legacy g:zip_* variables. Use old-zip when those features are required.

Only the original zip encryption is supported, not AES.

Legacy plugin: zip old-zip

The legacy Vimscript zip plugin browses, updates, and extracts archive entries. It is not loaded by default; use :packadd to activate it:

```
vim.cmd.packadd('old-zip')
```

This can be done at any time. Its manual is then available as :help pi_zip.txt.