learnbyexample

VIM REFERENCE GUIDE

Sundeep Agarwal

Table of contents

Preface

Vim Reference Guide is intended as a concise learning resource for beginner to intermediate level Vim users. It has more in common with cheatsheets than a typical text book. Most features are presented using a sample usage. Topics like Regular Expressions and Macros have more detailed explanations and examples due to their complexity.

The features covered in this guide are shaped and limited by my own experiences since 2007. You might expect me to have already become an expert, but I'm not there yet (nor do I have a pressing need for such expertise). The earlier version of this guide was written in 2017 and I did an extensive rework to get it fit for publication. A large portion of that time was spent correcting my understanding of Vim commands, going through user and reference manuals, getting good at using the built-in help, learning new features and so on.

Prerequisites

I do give a brief introduction to get started with using Vim, but having prior experience would be ideal before using this resource. As a minimum requirement, you should be able to use `vimtutor` on your own.

You are also expected to get comfortable with reading manuals, searching online, visiting external links provided for further reading, tinkering with the illustrated examples, asking for help when you are stuck and so on. In other words, be proactive and curious instead of just consuming the content passively.

See my Vim curated list for links to tutorials, books, interactive resources, cheatsheets, tips, tricks, forums and so on.

Conventions

- This guide is based on **Vim version 9.1** and some instructions assume Unix/Linux like operating systems. Where possible, details and resources are mentioned for other platforms.
- I prefer using **GVim**, so you might find some differences if you are using **Vim**.
- Built-in help command examples are also linked to an online version. For example, clicking :h usr_toc.txt will take you to table of contents for Vim User Manual. `:h usr_toc.txt` is also a command that you can use from within Vim.
- External links are provided throughout the book for exploring some topics in more depth.
- My vim_reference repo has markdown source and other details related to the book. If you are not familiar with the `git` command, click the **Code** button on the webpage to get the files.

How to use this guide

- Since many chapters take the form of cheatsheet with examples, this is a densely packed guide. Feel free to skim read some sections (because you already know them, not applicable for your use cases, etc), but try not to skip them entirely.
- If you are not able to understand a particular feature, go through the Vim user manual for that topic first. Each chapter has related documentation links at the top and external learning resources are often mentioned at the end of command descriptions.
- Practice the commands multiple times to build muscle memory.

- Building your own cheatsheet is highly recommended. You wouldn't need to refer most of the basic commands often, so you'll end up with a manageable reference sheet. As you continue to build muscle memory, you can prune the cheatsheet further.
- This guide covers a lot, but not everything. So, you'll need to learn from other resources too and add to your personal cheatsheet.

Acknowledgements

- Vim help files — user and reference manuals
- /r/vim/ and vi.stackexchange — helpful forums
- tex.stackexchange — for help on pandoc and `tex` related questions
- canva — cover image
- Warning and Info icons by Amada44 under public domain
- oxipng, pngquant and svgcleaner — for optimizing images
- Rodrigo Girão Serrão — for feedback and suggestions
- Andy — for cover image suggestions

Feedback and Errata

I would highly appreciate it if you'd let me know how you felt about this book. It could be anything from a simple thank you, pointing out a typo, mistakes in code snippets, which aspects of the book worked for you (or didn't!) and so on. Reader feedback is essential and especially so for self-published authors.

You can reach me via:

- Issue Manager: https://github.com/learnbyexample/vim_reference/issues
- E-mail: learnbyexample.net@gmail.com
- Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a lazy being who prefers to work just enough to support his modest lifestyle. He accumulated vast wealth working as a Design Engineer at Analog Devices and retired from the corporate world at the ripe age of twenty-eight. Unfortunately, he squandered his savings within a few years and had to scramble trying to earn a living. Against all odds, selling programming ebooks saved his lazy self from having to look for a job again. He can now afford all the fantasy ebooks he wants to read and spends unhealthy amount of time browsing the internet.

When the creative muse strikes, he can be found working on yet another programming ebook (which invariably ends up having at least one example with regular expressions). Researching materials for his ebooks and everyday social media usage drowned his bookmarks, so he maintains curated resource lists for sanity sake. He is thankful for free learning resources and open source tools. His own contributions can be found at https://github.com/learnbyexample.

List of books: https://learnbyexample.github.io/books/

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Code snippets are available under MIT License.

Resources mentioned in Acknowledgements section above are available under original licenses.

Book version

2.0

See Version_changes.md to track changes across book versions.

Introduction

Back in 2007, I had a rough beginning as a design engineer at a semiconductor company in terms of using software tools. Linux command line, Vim and Perl were all new to me. I distinctly remember progressing from `dd` (delete current line) to `d↓` (delete current line as well as the line below) and feeling happy that it reduced time spent on editing. Since I was learning on the job, I didn't know about count prefix or the various ways I could've deleted all the lines from the beginning of the file to the line containing a specific phrase. Or even better, I could've automated editing multiple files if I had been familiar with `sed` or progressed that far with Perl.

I also remember that we got a two-sided printed cheatsheet that we kept pinned to our cabins. That was one of the ways I kept adding commands to my repertoire. But, I didn't have a good insight to Vim's philosophy and I didn't know how to apply many of the cheatsheet commands. At some point, I decided to read the Vim book by Steve Oualline and that helped a lot, but it was also too long and comprehensive for me to read it all. My memory is hazy after that, and I don't recall what other resources I used. However, I'm sure I didn't effectively utilize built-in help. Nor did I know about stackoverflow or /r/vim until after I left my job in 2014.

Still, I knew enough to conduct a few Vim learning sessions for my colleagues. That came in handy when I got chances to teach Vim as part of a scripting course for college students. From 2016 to 2018, I started maintaining my tutorials on Linux command line, Vim and scripting languages as GitHub repos. As you might guess, I then started polishing these materials and published them as ebooks. This is an ongoing process, with **Vim Reference Guide** being the twelfth ebook.

Why Vim?

You've probably already heard that Vim is a text editor, powerful one at that. Vim's editing features feel like a programming language and you can customize the editor using scripting languages. Apart from a plethora of editing commands and support for regular expressions, you can also incorporate external commands. To sum it up, most editing tasks can be managed from within Vim itself instead of having to write a script.

Now, you might wonder, what is this need for complicated editing features? Why does a text editor require programming capabilities? Why is there even a requirement to *learn* how to use a text editor? Isn't it enough to have the ability to enter text, use keys like Backspace/Delete/Home/End/Arrow/etc, menubar, toolbar, some shortcuts, a search and replace feature and so on? A simple and short answer — to reduce repetitive manual task.

What I like the most about Vim:

- Lightweight and fast
- Modal editing helps me to think logically based on the type of editing task
- Composing commands and the ability to record them for future use
- Settings customization and creating new commands
- Integration with shell commands

There's a huge ecosystem of plugins, packages and colorschemes as well, but I haven't used them much. I've used Vim for a long time, but not really a power user. I prefer using GVim, tab pages, mouse, arrow keys, etc. So, if you come across tutorials and books suggesting you should avoid using them, remember that they are only subjective preferences.

Here are some more opinions by those who enjoy using Vim:

- stackoverflow: What are the benefits of learning Vim?
- Why Vim
- Vim Creep

 Should everybody use Vim? Is it suitable for all kinds of editing tasks? I'd say no. There are plenty of other well established text editors and new ones are coming up all the time. The learning curve isn't worth it for everybody. If Vim wasn't being used at job, I probably wouldn't have bothered with it. Don't use Vim for the wrong reasons article discusses this topic in more detail.

Installation

I use the following command on Ubuntu (a Linux distribution):

```
sudo apt install vim vim-gui-common
```

- :h usr_90.txt — user manual for installation on different platforms, common issues, upgrading, uninstallation, etc
- vi.stackexchange: How can I get a newer version of Vim? — building from source, using distribution packages, etc

 See https://github.com/vim/vim for source code and other details.

Ice Breaker

Open a terminal and follow these steps:

- `gvim ip.txt` opens a file named `ip.txt` for editing
  - You can also use `vim` if you prefer terminal instead of GUI, or if `gvim` is not available
- Press `i` key (yes, the lowercase alphabet `i`, not some alien key)
- Start typing, for example `What a weird editor`
- Press `Esc` key
- Press `:` key
- Type `wq`
- Press `Enter` key
- `cat ip.txt` — sanity check to see what you typed was saved or not

Phew, what a complicated procedure to write a simple line of text, isn't it? This is the most challenging and confusing part for a Vim newbie. Here's a brief explanation for the above steps:

- Vim is a **modal editor**. You have to be aware which mode you are in and use commands or type text accordingly
- When you first launch Vim, it starts in **Normal mode** (primarily used for editing and moving around)
- Pressing `i` key is one of the ways to enter **Insert mode** (where you type the text you want to save in a file)
- After you've entered the text, you need to save the file. To do so, you have to go back to Normal mode first by pressing the `Esc` key

- Then, you have to go to yet another mode! Pressing `:` key brings up the **Command-line mode** and awaits further instruction
- `wq` is a combination of **write** and **quit** commands
  - use `wq ip.txt` if you forgot to specify the filename while launching Vim, or perhaps if you opened Vim from the Start menu instead of a terminal
- `Enter` key completes the command you've typed

If you launched GVim, you'll likely have **Menu** and **Tool** bars, which would've helped with operations like saving, quitting, etc. Nothing wrong with using them, but this book will not discuss those operations. In fact, you'll learn how to configure Vim to hide them in the Customizing Vim chapter.

Don't proceed any further if you aren't comfortable with the above steps. Take help of youtube videos if you must. Master this basic procedure and you will be ready for Vim awesomeness that'll be discussed in the coming sections and chapters.

 Material presented here is based on GVim (GUI), which has a few subtle differences compared to Vim (TUI). See this stackoverflow thread for more details.

 Options and details related to opening Vim from the command line will be discussed in the CLI options chapter.

Built-in tutor

- `gvimtutor` command that opens a tutorial session with lessons to get started with Vim
  - don't worry if something goes wrong as you'll be working with a temporary file
  - use `vimtutor` if `gvim` is not available
  - **pro-tip**: go through this short tutorial multiple times, spread over multiple days and make copious notes for future reference

 Next step is :h usr_02.txt, which provides enough information about editing files with Vim.

 See also vimtutor-sequel, which provides advanced lessons.

Built-in help

Vim comes with comprehensive user and reference manuals. The user manual reads like a text book and reference manual has more details than you are likely to need. There's also an online site with these help contents, which will be linked as appropriate throughout this book.

- You can access built-in help in several ways:
  - type `:help` from Normal mode (or just the `:h` short form)
  - GVim has a `Help` menu
  - press `F1` key from Normal mode
- :h usr_toc.txt table of contents for User Manual

    - Task oriented explanations, from simple to complex. Reads from start to end like a book
- :h reference_toc table of contents for Reference Manual
    - Precise description of how everything in Vim works
- :h quickref quick reference guide
- :h help-summary effectively using help depending on the topic/feature you are interested in
    - See also vi.stackexchange: guideline to use help
- :h version9.txt what's new in Vim 9
    - See also VimLog, a ChangeLog for Vim

Here's a neat table from :h help-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 /[`

 You can go through a copy of the documentation online at https://vimhelp.org/. As shown above, all the `:h` hints in this book will also be linked to the appropriate online help section.

Vim learning resources

As mentioned in the Preface chapter, this **Vim Reference Guide** is more like a cheatsheet instead of a typical book for learning Vim. In addition to built-in features already mentioned in the previous sections, here are some resources you can use:

Tutorials

- Vim primer — learn Vim in a way that will stay with you for life
- Vim galore — everything you need to know about Vim
- Learn Vim progressively — short introduction that covers a lot
- Vim from the ground up — article series for beginners to expert users

Books

- Practical Vim
- Mastering Vim Quickly
- Learn Vim (the Smart Way)

Interactive

- OpenVim — interactive tutorial
- Vim Adventures — learn Vim by playing a game

- Learn vim and learn it fast — interactive lessons designed to help you get better at Vim faster

 See my Vim curated list for a more complete list of learning resources, cheatsheets, tips, tricks, forums, etc.

Modes of Operation

As mentioned earlier, Vim is a **modal editor**. This book will mainly discuss these four modes:

- Insert mode
- Normal mode
- Visual mode
- Command-line mode

This section provides a brief description for these modes. Separate chapters will discuss their features in more detail.

 For a complete list of modes, see :h vim-modes-intro and :h mode-switching. See also this comprehensive illustration of navigating modes.

Insert mode

This is the mode where the required text is typed. There are also commands available for moving around, deleting, autocompletion, etc.

Pressing the `Esc` key takes you back to the Normal mode.

Normal mode

This is the default mode when Vim is opened. This mode is used to run commands for operations like cut, copy, paste, recording, moving around, etc. This is also known as the Command mode.

Visual mode

Visual mode is used to edit text by selecting them first. Selection can either be done using mouse or visual commands.

Pressing the `Esc` key takes you back to the Normal mode.

Command-line mode

This mode is used to perform file operations like save, quit, search, replace, execute shell commands, etc. An operation is completed by pressing the `Enter` key after which the mode changes back to the Normal mode. The `Esc` key can be used to ignore whatever is typed and return to the Normal mode.

The space at the bottom of the screen used for this mode is referred to as Command-line area. It is usually a single line, but can expand for cases like auto completion, shell commands, etc.

Identifying the current mode

- In Insert mode, you get a blinking `|` cursor
    - also, `-- INSERT --` can be seen on the left hand side of the Command-line area

- In Normal mode, you get a blinking rectangular block cursor, something like this `█`
- In Visual mode, the Command-line area shows `-- VISUAL --` or `-- VISUAL LINE --` or `-- VISUAL BLOCK --` according to the visual command used
- In Command-line mode, the cursor is of course in the Command-line area


See also :h 'showmode' setting.

Vim philosophy and features


Commands discussed in this section will be covered again in later chapters. The idea here is to give you a brief introduction to modes and notable Vim features. See also:

- Best introduction to Vi and its core editing concepts explained as a language (this stackoverflow thread also has numerous Vim tips and tricks)
- Seven habits of effective text editing

As a programmer, I love how composable Vim commands are. For example, you can do this in Normal mode:

- `dG` delete from the current line to the end of the file
  - where `d` is the delete command awaiting further instruction
  - and `G` is a motion command to move to the last line of the file
- `yG` copy from the current line to the end of the file
  - where `y` is the yank (copy) command awaiting further instruction

Most Normal mode commands accept a count prefix. For example:

- `3p` paste the copied content three times
- `5x` delete the character under the cursor and 4 characters to its right (total 5 characters)
- `3` followed by `Ctrl` + `a` add `3` to the number under the cursor

There are context aware operations too. For example:

- `diw` delete a word regardless of where the cursor is on that word
- `ya}` copy all characters within `{}` including the `{}` characters

If you are a fan of selecting text before editing them, you can use the Visual mode. There are several commands you can use to start Visual mode. If enabled, you can even use mouse to select the required portions.

- `~` invert the case of the visually selected text (i.e. lowercase becomes UPPERCASE and vice versa)
- `g` followed by `Ctrl` + `a` for visually selected lines, increment numbers by `1` for the first line, by `2` for the second line, by `3` for the third line and so on

The Command-line mode is useful for file level operations, search and replace, changing Vim configurations, talking to external commands and so on.

- `/searchpattern` search the given pattern in the forward direction
- `:g/call/d` delete all lines containing `call`
- `:g/cat/ s/animal/mammal/g` replace `animal` with `mammal` only for the lines containing `cat`

- `:3,8! sort` sort only lines `3` to `8` (uses an external command `sort`)
- `:set incsearch` highlights the current match as you type the search pattern

Changes to Vim configurations from the Command-line mode are applicable only for that particular session. You can use the `vimrc` file to load the settings at startup.

- `colorscheme murphy` a dark theme
- `set tabstop=4` width for the tab character (default is `8`)
- `nnoremap <F5> :%y+<CR>` map `F5` key to copy everything to the system clipboard in Normal mode
- `inoreabbrev teh the` automatically correct `teh` to `the` in Insert mode

There are many more Vim features that'd help you with text processing and customizing the editor to your needs, some of which you'll get to know in the coming chapters.

Finally, you can apply your Vim skills elsewhere too. Vim-like features have been adopted across a huge variety of applications and plugins, for example:

- less command supports vim-like navigation
- Extensible vi layer for Emacs
- Vimium (browser extension), qutebrowser (keyboard-driven browser with vim-like navigation), etc
- JetBrains IdeaVim, VSCodeVim, etc
- Huge list of Vim-like applications and plugins

Vim's history

See Where Vim Came From if you are interested in knowing Vim's history that traces back to the 1960s with `qed`, `ed`, etc.

Chapters

Here's a list of remaining chapters:

Insert mode

This is the mode where the required text is typed. There are also commands available for moving around, deleting, autocompletion, etc.

Documentation links:

- :h usr_24.txt — overview of the most often used Insert mode commands
- :h insert.txt — reference manual for Insert and Replace mode


Recall that you need to add `i_` prefix for built-in help on Insert mode commands, for example :h i_CTRL-P.

Motion keys and commands

- `←` move left by one character within the current line
- `→` move right by one character within the current line
- `↓` move down by one line
- `↑` move up by one line
- `Ctrl` + `←` and `Ctrl` + `→` move to the start of the current/previous and next word respectively
  - From :h word "A word consists of a sequence of letters, digits and underscores, or a sequence of other non-blank characters, separated with white space"
  - you can also use `Shift` key instead of `Ctrl`
- `Home` move to the start of the line
- `End` move to the end of the line
- `PageUp` move up by one screen
- `PageDown` move down by one screen
- `Ctrl` + `Home` move to the start of the file
- `Ctrl` + `End` move to the end of the file


You can use the `whichwrap` setting (`ww` for short) to allow `←` and `→` arrow keys to cross lines. For example, `:set ww+=[,]` tells Vim to allow left and right arrow keys to move across lines in Insert mode (`+=` is used here to preserve existing options for the `whichwrap` setting).

Deleting

- `Delete` delete the character after the cursor
- `Backspace` delete the character before the cursor
  - `Ctrl` + `h` also deletes the character before the cursor
- `Ctrl` + `w` delete characters before the cursor until the start of a word
  - From :h word "A word consists of a sequence of letters, digits and underscores, or a sequence of other non-blank characters, separated with white space"
- `Ctrl` + `u` delete all the characters before the cursor in the current line, preserves indentation if any
  - if you have typed some characters in an existing line, this will delete characters till the starting point of the modification

Autocomplete word

- `Ctrl` + `p` autocomplete word based on matching words in the backward direction
- `Ctrl` + `n` autocomplete word based on matching words in the forward direction

If more than one word matches, they are displayed using a popup menu. You can take further action using the following options:

- `↑` and `↓` move up and down the list, but doesn't change the autocompleted text
- `Ctrl` + `p` and `Ctrl` + `n` move up and down the list as well as change the autocompleted text to that particular selection
- `Ctrl` + `y` confirm the current selection (the popup menu disappears)
    - you can also use the `Enter` key for confirmation if you have used the arrow keys to move through the popup list

 Typing any character will make the popup menu disappear and insert whatever character you had typed.

Autocomplete line

- `Ctrl` + `x` followed by `Ctrl` + `l` autocomplete line based on matching lines in the backward direction

 If more than one line matches, they are displayed using a popup menu. In addition to the options seen in the previous section, you can also use `Ctrl` + `l` to move up the list.

Autocomplete assist

- `Ctrl` + `e` cancels autocomplete
    - you'll retain the text you had typed before invoking autocomplete

 See :h ins-completion for more details and other autocomplete features. See :h 'complete' setting for customizing autocomplete commands.

Execute a Normal mode command

- `Ctrl` + `o` execute a Normal mode command and return to Insert mode
    - `Ctrl` + `o` followed by `A` moves the cursor to the end of the current line
    - `Ctrl` + `o` followed by `3j` moves the cursor three lines below
    - `Ctrl` + `o` followed by `ce` clear till the end of the word

Indenting

- `Ctrl` + `t` indent the current line
- `Ctrl` + `d` unindent the current line
- `0` followed by `Ctrl` + `d` deletes all indentation in the current line

 Cursor can be anywhere in the line for the above features. Indentation depends on the `shiftwidth` setting. See :h 'shiftwidth' for more details.

Insert register contents

- `Ctrl` + `r` helps to insert the contents of a register
  - `Ctrl` + `r` followed by `%` inserts the current file name
  - `Ctrl` + `r` followed by `a` inserts the content of `"a` register
- `Ctrl` + `r` followed by `=` allows you to insert the result of an expression
  - `Ctrl` + `r` followed by `=12+1012` and then `Enter` key inserts `1024`
  - `Ctrl` + `r` followed by `=strftime("%Y/%m/%d")` and then `Enter` key inserts the current date, for example `2022/02/02`

From :h 24.6:

> 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}` .

 Registers will be discussed in more details in the Normal mode chapter. See :h usr_41.txt to get started with Vim script.

Insert special characters

- `Ctrl` + `v` helps to insert special keys literally
  - `Ctrl` + `v` followed by `Esc` gives `^[`
  - `Ctrl` + `v` followed by `Enter` gives `^M`
- `Ctrl` + `q` alias for `Ctrl` + `v` , helps if it is mapped to do something else

 You'll see a practical usage of this command in the Macro chapter. You can also specify the character using decimal, octal or hexadecimal formats. See :h 24.8 for more details.

Insert digraphs

- `Ctrl` + `k` helps to insert digraphs (two character combinations used to represent a single character, such characters are usually not available on the keyboard)
  - `Ctrl` + `k` followed by `Ye` gives `¥`

 You can use `:digraphs` to get a list of combinations and their respective characters. You can also define your own combinations using the `:digraph` command. See :h 24.9 for more details.

Normal mode

Make sure you are in Normal mode before trying out the commands in this chapter. Press `Esc` key to return to Normal mode from other modes. Press `Esc` again if needed.

Documentation links:

- :h usr_03.txt — moving around
- :h usr_04.txt — making small changes
- :h motion.txt — reference manual for motion commands
- :h change.txt — reference manual for commands that delete or change text
- :h undo.txt — reference manual for undo and redo

Arrow motions

The four arrow keys can be used in Vim to move around, just like other text editors. Vim also maps them to four letters in Normal mode.

- `h` or `←` move left by one character within the current line
- `j` or `↓` move down by one line
- `k` or `↑` move up by one line
- `l` or `→` move right by one character within the current line

Vim offers a plethora of motion commands. Several sections will discuss them later in this chapter.

You can use the `whichwrap` setting to allow `←` and `→` arrow keys to cross lines. For example, `:set ww+=<,>` tells Vim to allow left and right arrow keys to move across lines in Normal and Visual modes. Add `h` and `l` to this comma separated list if want those commands to cross lines as well.

Cut

There are various ways to delete text. All of these commands can be prefixed with a **count** value. `d` and `c` commands can accept any of the motion commands. Only arrow motion examples are shown in this section, many more variations will be discussed later in this chapter.

- `dd` delete the current line
- `2dd` delete the current line and the line below it (total 2 lines)
  - `dj` or `d↓` can also be used
- `10dd` delete the current line and 9 lines below it (total 10 lines)
- `dk` delete the current line and the line above it
  - `d↑` can also be used
- `d3k` delete the current line and 3 lines above it (total 4 lines)
  - `3dk` can also be used
- `D` delete from the current character to the end of line (same as `d$`, where `$` is a motion command to move to the end of line)
- `x` delete only the current character under the cursor (same as `dl`)
- `5x` delete the character under the cursor and 4 characters to its right (total 5 characters)
- `X` delete the character before the cursor (same as `dh`)

    - if the cursor is on the first character in the line, deleting would depend on the `whichwrap` setting as discussed earlier
- `5X` delete 5 characters to the left of the cursor
- `cc` delete the current line and change to Insert mode
    - indentation will be preserved depending on the `autoindent` setting
- `4cc` delete the current line and 3 lines below it and change to Insert mode (total 4 lines)
- `C` delete from the current character to the end of line and change to Insert mode
- `s` delete only the character under the cursor and change to Insert mode (same as `cl`)
- `5s` delete the character under the cursor and 4 characters to its right and change to Insert mode (total 5 characters)
- `S` delete the current line and change to Insert mode (same as `cc`)
    - indentation will be preserved depending on the `autoindent` setting

 You can also select text (using mouse or visual commands) and then press `d` or `x` or `c` or `s` to delete the selected portions. Example usage will be discussed in the Visual mode chapter.

 The deleted portions can also be pasted elsewhere using the paste command (discussed later in this chapter).

Copy

There are various ways to copy text using the **yank** command `y` .

- `yy` copy the current line
    - `Y` also copies the current line
- `y$` copy from the current character to the end of line
    - use `:nnoremap Y y$` if you want `Y` to behave similarly to the `D` command
- `2yy` copy the current line and the line below it (total 2 lines)
    - `yj` and `y↓` can also be used
- `10yy` copy the current line and 9 lines below it (total 10 lines)
- `yk` copy the current line and the line above it
    - `y↑` can also be used

 You can also select text (using mouse or visual commands) and then press `y` to copy them.

Paste

The **put** (paste) command `p` is used after cut or copy operations.

- `p` paste the copied content once
    - if the copied text was line based, content is pasted **below** the current line
    - if the copied text was part of a line, content is pasted to the **right** of the cursor
- `P` paste the copied content once
    - if the copied text was line based, content is pasted **above** the current line

    - if the copied text was part of a line, content is pasted to the **left** of the cursor
- `3p` and `3P` paste the copied content three times
- `]p` paste the copied content like `p` command, but changes the indentation level to match the current line
- `[p` paste the copied content like `P` command, but changes the indentation level to match the current line

Undo

- `u` undo last change
    - press `u` again for further undos
- `U` undo latest changes on last edited line
    - press `U` again to redo changes


See :h 32.3 for details on `g-` and `g+` commands that you can use to undo branches.

Redo

- `Ctrl` + `r` redo a change undone by `u`
- `U` redo changes undone by `U`

Replace characters

Often, you just need to change one character. For example, changing `i` to `j`, `2` to `4` and so on.

- `rj` replace the character under the cursor with `j`
- `ry` replace the character under the cursor with `y`
- `3ra` replace the character under cursor as well as the two characters to the right with `aaa`
    - the command will entirely fail if there aren't sufficient characters to match the count

To replace multiple characters with different characters, use `R`.

- `Rlion` followed by `Esc` replace the character under cursor and three characters to the right with `lion`
    - `Esc` key marks the completion of `R` command
    - `Backspace` key will act as an undo command to give back the character that was replaced
    - if you are replacing at the end of a line, the line will be automatically extended if needed

The advantage of `r` and `R` commands is that you remain in the Normal mode, without needing to switch to Insert mode and back.

Repeat a change

- `.` the dot command repeats the last change
- If the last change was `2dd` (delete current line and the line below), dot key will repeat `2dd`

- If the last change was `5x` (delete current character and four characters to the right), dot key will repeat `5x`
- If the last change was `C123<Esc>` and dot key is pressed, it will clear from the current character to the end of the line, insert `123` and go back to Normal mode

From :h 4.3:

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


See :h repeat.txt for complex repeats, using Vim scripts, etc.

Open new line

- `o` open a new line below the current line and change to Insert mode
- `O` open a new line above the current line and change to Insert mode


Indentation of the new line depends on the `autoindent` , `smartindent` and `cindent` settings.

Moving within the current line

- `0` move to the beginning of the current line (i.e. column number 1)
  - you can also use the `Home` key
- `^` move to the beginning of the first non-blank character of the current line (useful for indented lines)
- `$` move to the end of the current line
  - you can also use the `End` key
  - `3$` move to the end of 2 lines below the current line
- `g_` move to the last non-blank character of the current line
- `3|` move to the third column character
  - `|` is same as `0` or `1|`

Moving within long lines that are spread over multiple screen lines:

- `g0` move to the beginning of the current screen line
- `g^` move to the first non-blank character of the current screen line
- `g$` move to the end of the current screen line
- `gj` move down by one screen line, prefix a count to move down by that many screen lines
- `gk` move up by one screen line, prefix a count to move up by that many screen lines
- `gm` move to the middle of the current screen line
  - **Note** that this is based on the screen width, not the number of characters in the line!
- `gM` move to the middle of the current line
  - **Note** that this is based on the total number of characters in the line


See :h left-right-motions for more details.

Character motions

These commands allow you to move based on a single character search, **within the current line only**.

- `f(` move forward to the next occurrence of character `(`
- `fb` move forward to the next occurrence of character `b`
- `3f"` move forward to the third occurrence of character `"`
- `t;` move forward to the character just before `;`
- `3tx` move forward to the character just before the third occurrence of character `x`
- `Fa` move backward to the character `a`
- `Ta` move backward to the character just after `a`
- `;` repeat the previous character motion in the same direction
- `,` repeat the previous character motion in the opposite direction
    - for example, `tc` becomes `Tc` and vice versa

Note that the previously used count prefix wouldn't be repeated with the `;` or `,` commands, but you can use a new count prefix. If you pressed a wrong motion command, use the `Esc` key to abandon the search instead of continuing with the wrongly chosen command.

Word motions

Definitions from :h word and :h WORD are quoted below:

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. 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.

- `w` move to the start of the next word
- `W` move to the start of the next WORD
    - `192.1.168.43;hello` is considered a single **WORD**, but has multiple **words**
- `b` move to the beginning of the current word if the cursor is *not* at the start of word. Otherwise, move to the beginning of the previous word
- `B` move to the beginning of the current WORD if the cursor is *not* at the start of WORD. Otherwise, move to the beginning of the previous WORD
- `e` move to the end of the current word if cursor is *not* at the end of word. Otherwise, move to the end of next word
- `E` move to the end of the current WORD if cursor is *not* at the end of WORD. Otherwise, move to the end of next WORD
- `ge` move to the end of the previous word
- `gE` move to the end of the previous WORD
- `3w` move 3 words forward
    - Similarly, a number can be prefixed for all the other commands mentioned above

 All of these motions will work across lines. For example, if the cursor is on the last word of a line, pressing `w` will move to the start of the first word in the next line.

Text object motions

- `(` move backward a sentence
- `)` move forward a sentence
- `{` move backward a paragraph
- `}` move forward a paragraph

 More such text objects will be discussed later under the Context editing section. See :h object-motions for a complete list of such motions.

Moving within the current file

- `gg` move to the first non-blank character of the first line
- `G` move to the first non-blank character of the last line
- `5G` move to the first non-blank character of the fifth line
  - As an alternative, you can use `:5` followed by `Enter` key (Command-line mode)
- `50%` move to halfway point
  - you can use other percentages as needed
- `%` move to matching pair of brackets like `()` , `{}` and `[]`
  - This will work across lines and nesting is taken into consideration as well
  - If the cursor is on a non-bracket character and a bracket character is present later in the line, the `%` command will move to the matching pair of that character (which could be present in some other line too)
  - Use the `matchpairs` option to customize the matching pairs. For example, `:set matchpairs+=<:>` will match `<>` as well

 It is also possible to match a pair of keywords like HTML tags, if-else, etc with `%` . See :h matchit-install for details.

Moving within the visible window

- `H` move to the first non-blank character of the top (home) line of the visible window
- `M` move to the first non-blank character of the middle line of the visible window
- `L` move to the first non-blank character of the bottom (low) line of the visible window

Scrolling

- `Ctrl` + `d` scroll half a page down
- `Ctrl` + `u` scroll half a page up
- `Ctrl` + `f` scroll one page forward
- `Ctrl` + `b` scroll one page backward
- `Ctrl` followed by **Mouse Scroll** scroll one page forward or backward

Reposition the current line

- `Ctrl` + `e` scroll up by a line
- `Ctrl` + `y` scroll down by a line
- `zz` reposition the current line to the middle of the visible window
  - useful to see context around lines that are nearer to the top/bottom of the visible window
- `zt` reposition the current line to the top of the visible window
- `zb` reposition the current line to the bottom of the visible window


See :h 'scrolloff' option if you want to always show context around the current line.

Indenting

- `>` and `<` indent commands, waits for motion commands similar to `d` or `y`
- `>>` indent the current line
- `3>>` indent the current line and two lines below (same as `>2j`)
- `>k` indent the current line and the line above
- `>}` indent till the end of the paragraph
- `<<` unindent the current line
- `5<<` unindent the current line and four lines below (same as `<4j`)
- `<2k` unindent the current line and two lines above
- `=` auto indent code, use motion commands to indicate the portion to be indented
  - `=4j` auto indent the current line and four lines below
  - `=ip` auto indent the current paragraph (you'll learn about `ip` later in the Context editing section)


Indentation depends on the `shiftwidth` setting. See :h shift-left-right, :h = and :h 'shiftwidth' for more details.


You can indent/unindent the same selection multiple times using a number prefix in the Visual mode.

Mark frequently used locations

- `ma` mark a location in the file using the alphabet `a`
  - you can use any of the 26 alphabets
  - use lowercase alphabets to work within the current file
  - use uppercase alphabets to work from any file
  - `:marks` will show a list of the existing marks
- `` `a `` move to the exact location marked by `a`
- `'a` move to the first non-blank character of the line marked by `a`
- `'A` move to the first non-blank character of the line marked by `A` (this will work for any file where the mark was set)
- `d`a` delete from the current character to the character marked by `a`
  - marks can be paired with any command that accept motions like `d` , `y` , `>` , etc

Motion commands that take you across lines (for example, `10G`) will automatically save the location you jumped from in the default `` ` `` mark. You can move back to that exact location using `` `` `` or the first non-blank character using `` '' ``. Note that the arrow and word motions aren't considered for the default mark even if they move across lines.

See :h mark-motions for more ways to use marks.

Jumping back and forth

This is helpful if you are moving around often while editing a large file, moving between different buffers, etc. From :h jump-motions:

> The following commands are **jump** commands: `'`, `` ` ``, `G`, `/`, `?`, `n`, `N`, `%`, `(`, `)`, `[[`, `]]`, `{`, `}`, `:s`, `:tag`, `L`, `M`, `H` and the commands that start editing a new file.
> 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.

- `Ctrl` + `o` navigate to the previous location in the jump list (`o` as in old)
- `Ctrl` + `i` navigate to the next location in the jump list (`i` and `o` are usually next to each other)
- `g;` go to the previous change location
- `g,` go to the newer change location
- `gi` place the cursor at the same position where it was left last time in the Insert mode

Use `:jumps` and `:changes` to view the jump and change lists respectively. See :h jump-motions for more details.

Edit with motion

From :h usr_03.txt:

> 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.

- `dG` delete from the current line to the end of the file
- `dgg` delete from the current line to the beginning of the file
- `d`a` delete from the current character up to the location marked by `a`
- `d%` delete up to the matching pairs for `()`, `{}`, `[]`, etc
- `ce` delete till the end of word and change to Insert mode
  - `cw` will also work the same as `ce`, this inconsistency is based on **Vi** behavior
  - use `:nnoremap cw dwi` if you don't want the old behavior
- `yl` copy the character under the cursor

- `yfc` copy from the character under the cursor up to the next occurrence of `c` in the same line
- `d)` delete up to the end of the sentence

From :h usr_03.txt:

> 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.

Context editing

You have seen examples for combining motions such as `w`, `%` and `f` with editing commands like `d`, `c` and `y`. Such combination of commands require precise positioning to be effective.

Vim also provides a list of handy context based options to make certain editing use cases easier using the `i` and `a` text object selections. You can easily remember the difference between these two options by thinking `i` as **inner** and `a` as **around**.

- `diw` delete a word regardless of where the cursor is on that word
  - Equivalent to using `de` when the cursor is on the first character of the word
- `diW` delete a WORD regardless of where the cursor is on that WORD
- `daw` delete a word regardless of where the cursor is on that word as well as a space character to the left/right of the word depending on its position in the current sentence
- `dis` delete a sentence regardless of where the cursor is on that sentence
- `yas` copy a sentence regardless of where the cursor is on that sentence as well as a space character to the left/right
- `cip` delete a paragraph regardless of where the cursor is on that paragraph and change to Insert mode
- `dit` delete all characters within HTML/XML tags, nesting is taken care as well
  - see :h tag-blocks for details about corner cases
- `di"` delete all characters within a pair of double quotes, regardless of where the cursor is within the quotes
- `da'` delete all characters within a pair of single quotes along with the quote characters
- `ci(` delete all characters within `()` and change to Insert mode
  - Works even if the parentheses are spread over multiple lines, nesting is taken care as well
- `ya}` copy all characters within `{}` including the `{}` characters
  - Works even if the braces are spread over multiple lines, nesting is taken care as well


You can use a count prefix for nested cases. For example, `c2i{` will clear the inner braces (including the braces, and this could be nested too) and then only the text between braces for the next level.


See :h text-objects for more details.

Named registers

You can use lowercase alphabets `a-z` to save some content for future use. You can also append some more content to those registers by using the corresponding uppercase alphabets `A-Z` at a later stage.

- `"ayy` copy the current line to the `"a` register
- `"Ayj` append the current line and the line below to the `"a` register
  - `"ayy` followed by `"Ayj` will result in total three lines in the `"a` register
- `"ap` paste content from the `"a` register
- `"eyiw` copy word under the cursor to the `"e` register


You can use `:reg` (short for `:registers`) to view the contents of the registers. Specifying one or more characters (next to each other as a single string) will display contents only for those registers.


The named registers are also used for saving macros (will be discussed in the Macro chapter). You can record an empty macro to clear the contents, for example `qbq` clears the `"b` register.

Special registers

Vim has nine other types of registers for different use cases. Here are some of them:

- `"` all yanked/deleted text is stored in this register
  - So, the `p` command is same as specifying `""p`
- `"0` yanked text is stored in this register
  - A possible use case: yank some content, delete something else and then paste the yanked content using `"0p`
- `"1` to `"9` deleted contents are stored in these registers and get shifted with each new deletion
  - `"1p` paste the contents of last deletion
  - `"2p` paste the contents of last but one deletion
- `"+` this register is used to work with the system clipboard contents
  - `gg"+yG` copy entire file contents to the clipboard
  - `"+p` paste content from the clipboard
- `"*` this register stores visually selected text
  - contents of this register can be pasted using **middle mouse button click** or `"*p` or `Shift` + `Insert`
- `"_` black hole register, when you want to delete something without saving it anywhere

Further reading

- :h registers
- stackoverflow: How to use Vim registers
- stackoverflow: Using registers on Command-line mode
- Advanced Vim registers

Search word nearest to the cursor

- `*` searches the word nearest to the cursor in the forward direction (matches only the whole word)
  - `Shift` followed by **left mouse click** can also be used in GVim
- `g*` searches the word nearest to the cursor in the forward direction (matches as part of another word as well)
  - for example, if you apply this command on the word `the`, you'll also get matches for `them`, `lather`, etc
- `#` searches the word nearest to the cursor in the backward direction (matches only the whole word)
- `g#` searches the word nearest to the cursor in the backward direction (matches as part of another word as well)


You can also provide a count prefix to these commands.

Join lines

- `J` joins the current line and the next line
  - the deleted `<EOL>` character is replaced with a space, unless there are trailing spaces or the next line starts with a `)` character
  - indentation from the lines being joined are removed, *except the current line*
- `3J` joins the current line and next two lines with one space in between the lines
- `gJ` joins the current line and the next line
  - `<EOL>` character is deleted (space character won't be added)
  - indentation won't be removed


`joinspaces`, `cpoptions` and `formatoptions` settings will affect the behavior of these commands. See :h J and scroll down for more details.

Changing case

- `~` invert the case of the character under the cursor (i.e. lowercase becomes UPPERCASE and vice versa)
- `g~` followed by motion command to invert the case of those characters
  - for example: `g~e`, `g~$`, `g~iw`, etc
- `gu` followed by motion command to change the case of those characters to lowercase
  - for example: `gue`, `gu$`, `guiw`, etc
- `gU` followed by motion command to change the case of those characters to UPPERCASE
  - for example: `gUe`, `gU$`, `gUiw`, etc


You can also provide a count prefix to these commands.

Increment and Decrement numbers

- `Ctrl` + `a` increment the number under the cursor or the first occurrence of a number to the right of the cursor
- `Ctrl` + `x` decrement the number under the cursor or the first occurrence of a number to the right of the cursor
- `3` followed by `Ctrl` + `a` adds `3` to the number
- `1000` followed by `Ctrl` + `x` subtracts `1000` from the number


Numbers prefixed with `0`, `0x` and `0b` will be treated as octal, hexadecimal and binary respectively (you can also use uppercase for `x` and `b`).


Decimal numbers prefixed with `-` will be treated as negative numbers. For example, using `Ctrl` + `a` on `-100` will give you `-99`. While this is handy, this trips me up often when dealing with date formats like 2021-12-07.

Miscellaneous

- `gf` opens a file using the path under the cursor
  - See :h gf and :h suffixesadd for more details
  - See :h window-tag if you want to open the file under the cursor as a split window, new tab and other usecases
- `Ctrl` + `g` display file information like name, number of lines, etc at the bottom of the screen
  - See :h CTRL-G for more details and related commands
- `g` followed by `Ctrl` + `g` display information about the current location of the cursor (column, line, word, character and byte counts)
- `ga` shows the codepoint value of the character under the cursor in decimal, octal and hexadecimal formats
- `g?` followed by motion command to change those characters with `rot13` transformation
  - `g?e` on start of `hello` word will change it to `uryyb`
  - `g?e` on start of `uryyb` word will change it to `hello`

Switching modes

Normal to Insert mode

- `i` place the cursor to the left of the current character (insert)
- `a` place the cursor to the right of the current character (append)
- `I` place the cursor before the first non-blank character of the line (helpful for indented lines)
- `gI` place the cursor before the first column of the line
- `gi` place the cursor at the same position where it was left last time in the Insert mode
- `A` place the cursor at the end of the line
- `o` open a new line below the current line and change to Insert mode
- `O` open a new line above the current line and change to Insert mode
- `s` delete the character under the cursor and change to Insert mode
- `S` delete the current line and change to Insert mode

    - `cc` can also be used
    - indentation will be preserved depending on the `autoindent` setting
- `C` delete from the current character to the end of line and change to Insert mode

Normal to Command-line mode

- `:` change to Command-line mode, awaits further commands
- `/` change to Command-line mode for searching in the forward direction
- `?` change to Command-line mode for searching in the backward direction

Normal to Visual mode

- `v` visually select the current character
- `V` visually select the current line
- `Ctrl` + `v` visually select column
- `gv` select previously highlighted visual area

 See :h mode-switching for a complete table. See also this comprehensive illustration of navigating modes.

Command-line mode

An operation in Command-line mode is completed by pressing the `Enter` key after which the mode changes back to the Normal mode. Press `Esc` key to ignore whatever is typed and return to the Normal mode.

Documentation links:

- :h usr_05.txt — set your settings
- :h usr_07.txt — editing more than one file
- :h usr_08.txt — splitting windows
- :h usr_10.txt — making big changes
- :h usr_21.txt — go away and come back
- :h 3.8 — simple searches
- :h cmdline.txt — reference manual for Command-line mode
- :h windows.txt — reference manual for Editing with multiple windows and buffers

Add `:` or `c_` prefix for built-in help on Command-line mode, for example :h :w and :h c_CTRL-R. Use single quotes around options, :h 'autoindent' for example.

Save changes

- `:w` save changes (`:w` is short for `:write`)
- `:w filename` provide a filename if it is a new file or if you want to save to another file
- `:w >> filename` append to an existing file (`w!` will create a new file if it doesn't exist)
- `:wa` save all the changed buffers (`:wa` is short for `:wall`)

Appending `!` forces Vim to override errors, provided you have the appropriate permissions. For example, if you have edited a read-only file, `:w` will result in an error and `:w!` will save the changes. Another case where you'll get an error is `:w filename` if the file already exists. Using `:w! filename` will override the error.

By default, the entire file content is used for these commands. You can use a range (discussed later in this chapter) to work with selective lines.

Quit Vim

- `:q` quit the current window (`:q` is short for `:quit`)
  - if other windows/tabs are present, they will remain open
  - you will get an error message if there are unsaved changes
- `:qa` quit all (`:qa` is short for `:quitall`)
  - you will get an error message if there are unsaved changes
- `:confirm qa` similar to quit all, but prompts for every file that has unsaved changes

Append `!` to discard unsaved changes and quit.

Combining Save and Quit

- `:wq` save changes and quit
- `:wqa` save changes for all files and quit


Append `!` to override errors. Not all errors can be skipped, for example unsaved changes on a file that hasn't been named yet.

Working with buffers and tabs

Multiple files can be opened in Vim within the same tab page and/or in different tabs. From :h windows-intro:

- A buffer is the in-memory text of a file.
- A window is a viewport on a buffer.
- A tab page is a collection of windows.

Buffers

- `:e` refreshes the current buffer (`:e` is short for `:edit`)
- `:e filename` open a particular file by its path, in the same window
- `:e #` switch back to the previous buffer, won't work if that buffer is not named
  - `:e#` can also be used (no space between `e` and `#`)
- `Ctrl + 6` switch back to the previous buffer, works even if that buffer is not named
  - `Ctrl + ^` can also be used
- `:e #1` open the first buffer
- `:e #2` open the second buffer, and so on
- `:buffers` show all buffers
  - `:ls` or `:files` can also be used
- `:bn` open the next file in the buffer list (`:bn` is short for `:bnext`)
  - opens the first buffer if you are on the last buffer
- `:bp` open the previous file in the buffer list (`:bp` is short for `:bprevious`)
  - opens the last buffer if you are on the first buffer


Use `:set hidden` if you want to switch to another buffer even if there are unsaved changes in the current buffer. Instead of this setting, you can also use `:hide edit filename` to hide the current unsaved buffer. You'll still get an error if you try to quit Vim without saving such buffers, unless you use the `!` modifier.


See :h 'autowrite' option if you want to automatically save changes when moving to another buffer.


See :h 22.4 and :h buffer-hidden for user and reference manuals on working with buffer lists.

Tabs

- `:tabe filename` open the given file in a new tab (`:tabe` is short for `:tabedit`)
  - if `filename` isn't specified, you'll get an unnamed empty window
  - by default, the new tab is opened to the right of the current tab
  - `:0tabe` open as the first tab
  - `:$tabe` open as the last tab
  - see :h :tabe for more details and features
- `:tabn` switch to the next tab (`:tabn` is short for `:tabnext`)
  - if tabs to the right are exhausted, switch to the first tab
  - `gt` and `Ctrl + Page Down` can also be used
  - `2gt` switch to the second tab (the number specified is absolute, not relative)
- `:tabp` switch to the previous tab (`:tabp` is short for `:tabprevious`)
  - if tabs to the left are exhausted, switch to the last tab
  - `gT` and `Ctrl + Page Up` can also be used
- `:tabr` switch to the first tab (`:tabr` is short for `:tabrewind`)
  - `:tabfirst` can also be used
- `:tabl` switch to the last tab (`:tabl` is short for `:tablast`)
- `:tabm N` move the current tab to after `N` tabs from the start (`:tabm` is short for `:tabmove`)
  - `:tabm 0` move the current tab to the beginning
  - `:tabm` move the current tab to the end
- `:tabm +N` move the current tab `N` positions to the right
- `:tabm -N` move the current tab `N` positions to the left


Buffer list includes all the files opened in all the tabs.


You can also use the mouse to switch/move tabs in GVim.

Splitting

- `:split filename` open file for editing in a new horizontal window, above the current window
  - you can also use `:sp` instead of `:split`
  - `:set splitbelow` open horizontal splits below the current window
- `:vsplit filename` open file for editing in a new vertical window, to the left of the current window
  - you can also use `:vs` instead of `:vsplit`
  - `:set splitright` open vertical splits to the right of the current window
- `Ctrl + w` followed by `w` switch to the below/right window for horizontal/vertical splits respectively
  - `Ctrl + w` followed by `Ctrl + w` also performs the same function
  - switches to the first split if you are on the last split
- `Ctrl + w` followed by `W` switch to the above/left window for horizontal/vertical splits respectively
  - switches to the last split if you are on the first split
- `Ctrl + w` followed by `hjkl` or arrow keys, switch in the respective direction

- `Ctrl` + `w` followed by `t` or `b` switch to the top (first) or bottom (last) window
- `Ctrl` + `w` followed by `HJKL` (uppercase), moves the current split to the farthest possible location in the respective direction


If filename is not provided, the current one is used.


Vim adds a highlighted horizontal bar containing the filename for each split.

Edit all buffers

If multiple buffers are open and you want to apply a common editing task for all of them, one option is to use the `bufdo` command.

- `:silent! bufdo %s/search/replace/g | update` perform substitution across all the buffers
  - `silent` skips displaying normal messages
  - `!` skips error messages
- It is not an efficient way to open buffers just to search and replace a pattern across multiple files. Use tools like `sed`, `awk` and `perl` instead.
  - See my book list if you'd like to learn about such tools.
- See :h :bufdo, :h :windo and :h :silent for more details.

Further reading

- How to change multiple files
- stackoverflow: Effectively work with multiple files
- When to use Buffers and when to use Tabs
- :h argument-list — for working only with the files provided as Vim CLI arguments

Setting options

From :h options.txt:

> 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
> - **number** has a numeric value
> - **string** has a string value

Here are examples for each of these forms:

- `:set cursorline` highlight the line containing the cursor
- `:set history=200` increase default history from 50 to 200
- `:set ww+=[,]` allow left and right arrow keys to move across lines in Insert mode
  - `+=` allows you to append to an existing string value

Usage guidelines:

- `set {option}` switch on the given boolean setting
  - `:set expandtab` use spaces for tab expansion

- `set {option}!` toggle the given boolean setting
  - `:set expandtab!` if previously tabs were expanded, it will be turned off and vice versa
  - `set inv{option}` can also be used
- `set no{option}` switch off the given boolean setting
  - `:set noexpandtab` disable expanding tab to spaces
- `set {option}?` get the current value of the given option (works for all three forms)
  - `:set expandtab?` output will be `expandtab` or `noexpandtab` depending on whether it is switched on or off
- `set {option}` get the current value of number or string option
  - for example, try `:set history` or `:set ww`


See :h options.txt for a complete list of usage guidelines and available options.

Search

- `/searchpattern` search the given pattern in the forward direction
- `?searchpattern` search the given pattern in the backward direction
- `Esc` ignore the currently typed pattern and return to Normal mode
- `n` move to the next match in the same direction as the last search
  - if you used `/` for searching, `n` will move in the forward direction
  - if you used `?` for searching, `n` will move in the backward direction
- `N` move to the next match in the opposite direction of the last search
- `/` followed by `Enter` repeat the last search in the forward direction
- `?` followed by `Enter` repeat the last search in the backward direction
- `Ctrl` + `c` cancel the search if it is taking too long

By default, the cursor is placed at the starting character of the match. There are various options to place the cursor at other locations:

- `/searchpattern/s` place the cursor at the start of the match
  - same as `/searchpattern` or `/searchpattern/s+0`
  - you can also use `b` (begin) instead of `s` , but it'll change to `s` after the command is executed
- `/searchpattern/s+2` place the cursor `2` characters after the start of the match (i.e. the third character of the match)
- `/searchpattern/s-2` place the cursor `2` characters before the start of the match
- `/searchpattern/e` place the cursor at the end of the match
- `/searchpattern/e+4` place the cursor `4` characters after the end of the match
- `/searchpattern/e-4` place the cursor `4` characters before the end of the match
- `/searchpattern/+3` place the cursor `3` lines below the match
- `/searchpattern/-3` place the cursor `3` lines above the match

Highlight settings:

- `:set hlsearch` highlight the matched patterns
- `:set nohlsearch` do not highlight matched patterns
- `:set hlsearch!` toggle the highlight setting
- `:set hlsearch?` check what is the current highlight setting

- `:set incsearch` highlights current match as you type the pattern, the screen is updated automatically as needed
  - other matching portions will be highlighted based on `hlsearch` settings
  - if you press `Esc` instead of `Enter`, you'll end up where you originally started before the search
- `:noh` clear the currently highlighted patterns, if any (`:noh` is short for `:nohlsearch`)


Using an empty pattern will repeat the last searched pattern. So, you can use something like `//s+3` to repeat the last search with this new offset. Empty pattern can be used with substitution command as well (discussed later in this chapter). See :h last-pattern for more details.


You can prefix a count value to the `/`, `?`, `n` and `N` commands. Also, searching will automatically wrap around when it reaches the top or bottom of the file contents, unless you set the `nowrapscan` option.


Characters like `.`, `^`, `$`, etc have special meaning in the `searchpattern`. These will be discussed in detail in the Regular Expressions chapter.

Range

By default, certain commands like `:d` and `:s` apply to the current line whereas commands like `:w` and `:perldo` apply to the entire file. You can prefix a range to change the lines that are acted upon.

- `:d` delete the current line (`:d` is short for the `:delete` command)
- `:.w filename` save the current line (represented by `.`) to the given filename
  - recall that by default `:w` works on the entire file
- `:5d` delete the fifth line
- `:$d` delete the last line (represented by `$`)
- `:25m0` move the twenty-fifth line to the start of the file (`:m` is short for the `:move` command)
  - the number following `m` is the line number *after* which you want to place the lines specified by the range
  - use `:t` (or `:co` or `:copy`) command if you want to copy instead of moving
- `:2,10d` delete the second to tenth lines (comma is used here to separate the start and end ranges)
- `:5,$d` delete the fifth line to the last line
- `:5,$-1d` delete the fifth line to the last but one line
- `:%d` delete all the lines (`%` is a shortcut for the `1,$` range)
- `:/pat1/,/pat2/d` delete the matching range of lines in the forward direction from the current cursor position (forward because `/` is used)
  - if there are multiple matches, only the first such range will be deleted
  - use `?pattern?` to find a match in the backward direction
  - you can also mix these two types of direction if needed
- `:/pat1/;+1d` delete the line matching `pat1` as well as the line after (total `2` lines)

    - using `;` will set the line matched by the first pair of the range as the current line for the second pair
- `:/pat1/;-2d` delete the line matching `pat1` as well as two lines before (total `3` lines)
- `:5;/pat1/d` delete from the fifth line to a line matching `pat1` after the fifth line
    - note the use of `;` again here, the search will be based on the current cursor line if you use `,` instead of `;`
- `:'a,'bd` delete from the line marked by `a` to the line marked by `b`


If you press `:` after a visual selection, you'll automatically get `:'<,'>` as the visual range. If you prefix a number before pressing `:`, you'll get a range with that many lines — for example `10:` will give you `:.,.+9` as the range.


See :h 10.3 and :h cmdline-ranges for more details.


See :h ex-cmd-index for a complete list of `:` commands.

Search and Replace

```
:[range]s[ubstitute]/{pattern}/{string}/[flags] [count]
```

General syntax for the `s` command (short for `substitute`) is shown above. Space between the `range` and `s` is optional, which you can use for readability.

- `: s/a/b/` replace the first occurrence of `a` with `b` on the current line only
    - you can also use `:. s/a/b/` (recall that `.` represents the current line)
    - the delimiter after the replacement string is optional in this case
- `:2 s/apple/Mango/i` replace the first occurrence of `apple` with `Mango` on the second line only
    - `i` flag matches the `searchpattern` case insensitively
- `:3,6 s/call/jump/g` replace all the occurrences of `call` with `jump` on lines `3` to `6`
    - `g` flag performs search and replace for all the matching occurrences
- `:5,$ s/call/jump/g` replace all the occurrences of `call` with `jump` from the fifth line to the end of the file
- `:% s/call/jump/g` replace all the occurrences of `call` in the file with `jump`
    - recall that `%` is a shortcut for the range `1,$`


You can leave the `searchpattern` as empty to reuse the previously searched pattern, which could be from `/`, `?`, `*`, `s` command, etc. See :h last-pattern for more details.


See the Regular Expressions chapter for more details on the substitute command.

Editing lines filtered by a pattern

```
:[range]g[lobal]/{pattern}/[cmd]
```

General syntax for the `g` command (short for `global`) is shown above. This command is used to edit lines that are first filtered based on a `searchpattern`. You can use `g!` or `v` to act on lines *not* satisfying the filtering condition.

- `:g/call/d` delete all lines containing `call`
  - similar to the `d` Normal mode command, the deleted contents will be saved to the default `"` register
  - `:g/call/d a` in addition to the default register, the deleted content will also be stored in the `"a` register
  - `:g/call/d _` deleted content won't be saved anywhere, since it uses the black hole register
- `:g/^#/t0` copy all lines starting with `#` to the start of the file
- `:1,5 g/call/d` delete all lines containing `call` only for the first five lines
- `:v/jump/d` delete all lines *not* containing `jump`
  - same as `:g!/jump/d`
- `:g/cat/ s/animal/mammal/g` replace `animal` with `mammal` only for the lines containing `cat`
- `:.,.+20 g/^#/ normal >>` indent the current line and the next `20` lines only if the line starts with `#`
  - Note the use of `normal` when you need to use Normal mode commands on the filtered lines
  - Use `normal!` if you don't want user defined mappings to be considered


In addition to the `/` delimiter, you can also use any single byte character other than alphabets, `\`, `"` or `|`.


See :h :g for more details.

Shell commands

You can also use shell commands from within Vim (assuming you have access to these commands).

- `:!ls` execute the given shell command and display its output
  - the results are displayed as part of an expanded Command-line area, doesn't change contents of the file
- `:.! date` replace the current line with the output of the given command
  - pressing `!!` in Normal mode will also result in `:.!`
  - `!` waits for motion similar to the `d` and `y` commands, `!G` will give `:.,$!`
- `:%! sort` sort all the lines
  - recall that `%` is a shortcut for the range `1,$`
  - note that this executes an external command, not the built-in `:sort` command
- `:3,8! sort` sort only lines `3` to `8`

- `:r! date` insert output of the given command below the current line
- `:r report.log` insert contents of the given file below the current line
  - Note that `!` is not used here since there is no shell command
- `:.!grep '^Help ' %` replace the current line with all the lines starting with `Help` in the current file
  - `%` here refers to the current file contents
- `:sh` open a shell session within Vim
  - use the `exit` command to quit the session

 See :h :!, :h :sh and :h :r for more details.

Terminal mode

- `:terminal` open a new terminal window as a horizontal split
  - the terminal window opens above the current window unless `splitbelow` option is set
  - you can then use shell commands as you would normally do from a terminal
- `:vertical :terminal` open a new terminal window as a vertical split
  - the terminal window opens to the left of the current window unless `splitright` option is set
- `Ctrl` + `w` followed by `w` or `Ctrl` + `w` move to the next window
  - helps you to easily switch back and forth if you have one text editing window and one terminal window
  - see the Splitting section discussed earlier in this chapter for more such commands
- `Ctrl` + `w` followed by `N` goes to Terminal-Normal mode which will help you to move around using Normal mode commands, copy text, etc (note that you need to use uppercase `N` here)
  - `Ctrl` + `\` followed by `Ctrl` + `n` another way to go to Terminal-Normal mode
  - `:tnoremap <Esc> <C-w>N` map `Esc` key to go to Terminal-Normal mode (use of maps will be discussed in more detail in the Customizing Vim chapter)
- `Ctrl` + `w` followed by `:` go to Command-line mode from terminal window

 Depending on your shell, you can use the `exit` command to end the terminal session. `Ctrl+d` might work too.

 There are lot of features in this mode, see :h terminal.txt for more details.

Line number settings

- `:set number` prefix line numbers
  - this is a visual guideline, doesn't modify the text
  - see :h 'numberwidth' for setting the width of the prefix
- `:set number!` toggle number setting
- `:set nonumber` don't use line number prefix
- `:set relativenumber` prefix line numbers relative to the current line

    - current line is assigned `0`, lines above and below the current line are assigned `1`, two lines above and below are assigned `2` and so on
    - useful visual guide for commands like `11yy`, `6>>`, `9j`, etc
- `:set relativenumber!` toggle relative number setting
- `:set norelativenumber` don't use relative line number prefix


See :h 5.9 for user manual about often used options.

Sessions

- `:mksession proj.vim` save the current Vim session with details like cursor position, file list, layout, etc
    - you can customize things to be saved using the `sessionoptions` setting
    - for example, `:set sessionoptions+=resize` will save resized window information as well
- `:mksession! proj.vim` overwrite existing session
- `:source proj.vim` restore Vim session from `proj.vim` file
    - `vim -S proj.vim` restore a session from the command line when launching Vim


See :h 21.4, :h views-sessions and :h 'sessionoptions' for more details.


See stackoverflow: How to save and restore multiple different sessions in Vim? for custom settings to automate the save and restore process and other tips and tricks. See also Learn-Vim: Views, Sessions, and Viminfo.

Viminfo

From :h 21.3:

> 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 `viminfo` 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 `viminfo` file. When Vim starts again, the `viminfo` file is read and the information restored.

The `:mksession` command doesn't save the `viminfo` file. You'll have to save and restore this file separately:

- `:wviminfo! proj.viminfo` save the current internal Viminfo contents to the given file
    - if `!` isn't used, you'll get a merged output based on the current internal Viminfo contents and the file contents

- `:rviminfo! proj.viminfo` restore Viminfo from `proj.viminfo` file
  - `!` overwrites any existing internal settings
  - `vim -i proj.viminfo` restore Viminfo from the command line when launching Vim


See :h viminfo-read-write for more details.

Motion, editing and completion commands

Once you are in Command-line mode (after typing `:` or `/` or `?`), you can use the commands discussed below. Many of these commands are similar to those available in the Insert mode.

- `←` and `→` move the cursor left and right respectively by one character
  - you can also use the mouse to position the cursor
- `Ctrl` + `←` and `Ctrl` + `→` move the cursor left and right respectively by one WORD
  - you can also use the `Shift` key instead of `Ctrl`
  - Note that in Insert mode this command moves by word, not WORD
- `Ctrl` + `b` or `Home` move to the beginning
- `Ctrl` + `e` or `End` move to the end
- `Ctrl` + `w` delete word before the cursor
- `Ctrl` + `u` delete all characters before the cursor
- `Ctrl` + `r` insert register contents
  - `Ctrl` + `r` followed by `%` inserts the current file name
  - `Ctrl` + `r` followed by `a` inserts the content of `"a` register
- `Ctrl` + `r` followed by `=` allows you to insert the result of an expression
  - `Ctrl` + `r` followed by `=12+1012` and then `Enter` key inserts `1024`
  - `Ctrl` + `r` followed by `=strftime("%Y/%m/%d")` and then `Enter` key inserts the current date, for example `2022/02/02`
- `Ctrl` + `d` show completions based on the characters typed so far
- `Tab` autocomplete based on the characters typed so far, pressing this key multiple times will cycle through the completions
  - behavior can be customized using the `wildmode` setting
- `Ctrl` + `c` cancel and go back to Normal mode
- `Esc` cancel and go back to Normal mode, depends on `cpoptions` setting


See :h usr_20.txt for a nice tutorial on working effectively in the Command-line mode.
See :h cmdline-editing and :h cmdline-completion for more details.

Command-line history

There are separate history lists for `:` commands, `searchpattern` , etc. These lists make it easy to reuse (after modifications if necessary) previously executed commands.

- `↑` and `↓` move through the history lists
  - if you have already typed some characters, you will get only the commands starting with those characters

 See :h cmdline-history for more details. You can change the number of entries that are remembered using the `history` setting.

Command-line window

You can also view, edit and execute the history of commands using a special Command-line window. You can open this special window from Normal mode as well as Command-line mode. This window will be in Normal mode by default, which will allow you to move around easily. You can also edit any of the history commands if you wish.

- `q:` window for `:` commands (from Normal mode)
- `q/` and `q?` window for search patterns (from Normal mode)
- `Ctrl` + `f` use this shortcut if you are already in the Command-line mode, opens appropriate `:` or search pattern windows automatically, any text you've typed so far will be preserved as the most recent command
- `Enter` to execute the command under the cursor
- `Ctrl` + `c` continue editing the command under the cursor in the usual Command-line area, the window will still be visible
- `:q` quit the window and go to Normal mode

 See :h cmdline-window for more details. You can change the number of entries that are remembered using the `history` setting.

Visual mode

Visual mode allows you to perform editing tasks on selected portions of text. There are various visual commands to select the text of interest. If enabled, you can also use your mouse to select the desired portions.

Documentation links:

- :h 4.4 — visual mode
- :h 10.5 — visual block mode
- :h visual.txt — reference manual for visual mode

Recall that you need to add `v_` prefix for built-in help on Visual mode commands, for example :h v_o.

Selection

- `v` visually select the current character, use any motion command to extend the selection
  - `ve` selects till the end of a word
  - `vip` selects a paragraph (text object) and so on
- `V` visually select the current line, you can extend this using motion commands
  - `Vgg` selects the current line and then extends to the start of the file
- `Ctrl` + `v` visually select columns
  - `Ctrl` + `v` followed by `3j2l` selects a 4x3 block
  - you can also select using the mouse and then press `Ctrl` + `v`
- `gv` select previously highlighted visual area
- `o` move the cursor to the diagonally opposite corner of the visual selection
- `O` move the cursor to the other corner of the current line in visual block selection
- `Esc` go back to Normal mode

Pressing `$` in block selection will select until the end of lines for the selected area, even if the lines have different number of characters. This will continue to be the case if you extend the selection with up/down motions.

Editing

- `d` delete the selected text
- `y` yank (copy) the selected text
- `p` replace the selected text with contents of the default `"` register
  - See :h v_p for more details, corner cases, uppercase `P` command behavior, etc
- `c` clear the selected text and change to Insert mode
  - for block selection, you can type some text and press the `Esc` key to replicate that text across all the visually selected lines
- `C` similar to `c` but clears till the end of lines before changing to Insert mode
- `I` for block selection, press `I`, type some text and press the `Esc` key to replicate that text across all lines to the left of the column
  - text will not be inserted for lines shorter than the starting column of selection
  - if you type a multiline text, it will not get replicated

- `A` for block selection, press `A`, type some text and press the `Esc` key to replicate that text across all lines to the right of the column
  - if `$` was used for selection, text will be inserted only after the end of respective lines
  - otherwise, text will be inserted after the selected column and space characters will used to extend shorter lines if any
  - if you type a multiline text, it will not get replicated
- `ra` replace every character of the selected text with `a`
- `:` perform Command-line mode editing commands like `g`, `s`, `!`, `normal`, etc on the selected text
- `J` and `gJ` join lines using the same rules as seen in Normal mode


Press `Ctrl + c` if you've typed text after using `I` or `A` but don't want to replicate the text across all the lines.


See :h visual-operators for a complete list of commands.

Search and Select

- `gn` search the last used pattern in the forward direction and visually select the matched portion
  - selects the current match if the cursor is anywhere within the matching portion
  - extends the visual selection if Visual mode is already active
- `gN` search the last used pattern in the backward direction and visually select the matched portion
- `cgn` here `gn` acts as the motion for the Normal mode command `c`
  - since this is considered as a single change, pressing `.` will change the next match in the forward direction
  - whereas, if you apply `c` with Normal mode motion, you'll have to first use `n` (or `N` depending on the direction) and then use the `.` command to repeat the change


Searching will automatically wrap around when it reaches the top or bottom of the file contents, unless you set the `nowrapscan` option.

Indenting

- `>` indent the visual selection once
- `3>` indent the visual selection three times
- `<` unindent the visual selection once
- `=` auto indent code

Consider the following unindented code:

```
for(i=1; i<5; i++)
{
for(j=i; j<10; j++)
{
```

```
statements
}
statements
}
```

Here's the result after applying `vip=` (you can also use `=ip` if you prefer Normal mode).

```
for(i=1; i<5; i++)
{
    for(j=i; j<10; j++)
    {
        statements
    }
    statements
}
```

 For block selection, space will be inserted before the starting column of the block.

 Indentation depends on the `shiftwidth` setting. See :h shift-left-right, :h = and :h 'shiftwidth' for more details.

Changing Case

- `~` invert the case of the visually selected text (i.e. lowercase becomes UPPERCASE and vice versa)
- `U` change the visually selected text to UPPERCASE
- `u` change the visually selected text to lowercase

Increment and Decrement numbers

- `Ctrl` + `a` increment by `1`
- `5` followed by `Ctrl` + `a` increment by `5`
- `Ctrl` + `x` decrement by `1`
- `g` followed by `Ctrl` + `a` increment by `1` for the first line, by `2` for the second line, by `3` for the third line and so on
- `2g` followed by `Ctrl` + `a` increment by `2` for the first line, by `4` for the second line, by `6` for the third line and so on (i.e. repeat the process specified by the count prefix)
- `g` followed by `Ctrl` + `x` decrement by `1` for the first line, by `2` for the second line, by `3` for the third line and so on

 The visual selection should cover the numeric portion you wish to increment or decrement. If there are multiple numbers in a visually selected line, only the first number will be affected.

Example for `g` followed by `Ctrl` + `a` :

```
# before
item[0]
item[0]
item[0]

# after
item[1]
item[2]
item[3]
```

Example for `g` followed by `Ctrl` + `x` :

```
# before
item[12]
item[16]
item[22]

# after
item[11]
item[14]
item[19]
```

Example for `3g` followed by `Ctrl` + `a` :

```
# before
item[12]
item[16]
item[22]

# after
item[15]
item[22]
item[31]
```

Regular Expressions

This chapter will discuss regular expressions (regexp) and related features in detail. As discussed in earlier chapters:

- `/searchpattern` search the given pattern in the forward direction
- `?searchpattern` search the given pattern in the backward direction
- `:range s/searchpattern/replacestring/flags` search and replace
  - `:s` is short for the `:substitute` command
  - the delimiter after the `replacestring` portion is optional if you are not using flags

Documentation links:

- :h usr_27.txt — search commands and patterns
- :h pattern-searches — reference manual for Patterns and search commands
- :h :substitute — reference manual for the `:substitute` command

Recall that you need to add the `/` prefix for built-in help on regular expressions, :h /^ for example.

Flags

- `g` replace all occurrences within a matching line
  - by default, only the first matching portion will be replaced
- `c` ask for confirmation before each replacement
- `i` ignore case for `searchpattern`
- `I` don't ignore case for `searchpattern`

These flags are applicable for the substitute command but not the `/` or `?` searches. Flags can also be combined, for example:

- `s/cat/Dog/gi` replace every occurrence of `cat` with `Dog`
  - Case is ignored, so `Cat`, `cAt`, `CAT`, etc will also be replaced
  - Note that `i` doesn't affect the case of the replacement string

See :h s_flags for a complete list of flags and more details about them.

Anchors

By default, regexp will match anywhere in the text. You can use line and word anchors to specify additional restrictions regarding the position of matches. These restrictions are made possible by assigning special meaning to certain characters and escape sequences. The characters with special meaning are known as **metacharacters** in regular expressions parlance. In case you need to match those characters literally, you need to escape them with a `\` character (discussed in the Escaping metacharacters section later in this chapter).

- `^` restricts the match to the start-of-line
  - `^This` matches `This is a sample` but not `Do This`
- `$` restricts the match to the end-of-line

    - `)$` matches `apple (5)` but not `def greeting():`
- `^$` match empty lines
- `\<pattern` restricts the match to the start of a word
    - word characters include alphabets, digits and underscore
    - `\<his` matches `his` or `to-his` or `history` but not `this` or `_hist`
- `pattern\>` restricts the match to the end of a word
    - `his\>` matches `his` or `to-his` or `this` but not `history` or `_hist`
- `\<pattern\>` restricts the match between the start of a word and end of a word
    - `\<his\>` matches `his` or `to-his` but not `this` or `history` or `_hist`


End-of-line can be `\r` (carriage return), `\n` (newline) or `\r\n` depending on your operating system and the `fileformat` setting.


See :h pattern-atoms for more details.

Dot metacharacter

- `.` match any single character other than end-of-line
    - `c.t` matches `cat` or `cot` or `c2t` or `c^t` or `c.t` or `c;t` but not `cant` or `act` or `sit`
- `\_.` match any single character, including end-of-line


As seen above, matching end-of-line character requires special attention. Which is why examples and descriptions in this chapter will assume you are operating line wise unless otherwise mentioned. You'll later see how `\_` is used in many more places to include end-of-line in the matches.

Greedy Quantifiers

Quantifiers can be applied to literal characters, the dot metacharacter, groups, backreferences and character classes. Basic examples are shown below, more will be discussed in the sections to follow.

- `*` match zero or more times
    - `abc*` matches `ab` or `abc` or `abccc` or `abccccc` but not `bc`
    - `Error.*valid` matches `Error: invalid input` but not `valid Error`
    - `s/a.*b/X/` replaces `table bottle bus` with `tXus`
- `\+` match one or more times
    - `abc\+` matches `abc` or `abccc` but not `ab` or `bc`
- `\?` match zero or one times
    - `\=` can also be used, helpful if you are searching backwards with the `?` command
    - `abc\?` matches `ab` or `abc` . This will match `abccc` or `abccccc` as well, but only the `abc` portion
    - `s/abc\?/X/` replaces `abcc` with `Xc`
- `\{m,n}` match `m` to `n` times (inclusive)

    - `ab\{1,4}c` matches `abc` or `abbc` or `xabbbcz` but not `ac` or `abbbbbc`
    - if you are familiar with BRE, you can also use `\{m,n\}` (ending brace is escaped)
- `\{m,}` match at least `m` times
    - `ab\{3,}c` matches `xabbbcz` or `abbbbbc` but not `ac` or `abc` or `abbc`
- `\{,n}` match up to `n` times (including `0` times)
    - `ab\{,2}c` matches `abc` or `ac` or `abbc` but not `xabbbcz` or `abbbbbc`
- `\{n}` match exactly `n` times
    - `ab\{3}c` matches `xabbbcz` but not `abbc` or `abbbbbc`

Greedy quantifiers will consume as *much* as possible, provided the overall pattern is also matched. That's how the `Error.*valid` example worked. If `.*` had consumed everything after `Error` , there wouldn't be any more characters to try to match `valid` . How the regexp engine handles matching varying amount of characters depends on the implementation details (backtracking, NFA, etc).


See :h pattern-overview for more details.


If you are familiar with other regular expression flavors like Perl, Python, etc, you'd be surprised by the use of `\` in the above examples. If you use the `\v` very magic modifier (discussed later in this chapter), the `\` won't be needed.

Non-greedy Quantifiers

Non-greedy quantifiers match as *minimally* as possible, provided the overall pattern is also matched.

- `\{-}` match zero or more times as minimally as possible
    - `s/t.\{-}a/X/g` replaces `that is quite a fabricated tale` with `XX fabricaXle`
        - ⋆ the matching portions are `tha` , `t is quite a` and `ted ta`
    - `s/t.*a/X/g` replaces `that is quite a fabricated tale` with `Xle` since `*` is greedy
- `\{-m,n}` match `m` to `n` times as minimally as possible
    - `m` or `n` can be left out as seen in the previous section
    - `s/.\{-2,5}/X/` replaces `123456789` with `X3456789` (here `.` matched 2 times)
    - `s/.\{-2,5}6/X/` replaces `123456789` with `X789` (here `.` matched 5 times)


See :h pattern-overview and stackoverflow: non-greedy matching for more details.

Character Classes

To create a custom placeholder for a limited set of characters, you can enclose them inside the `[]` metacharacters. Character classes have their own versions of metacharacters and provide special predefined sets for common use cases.

- `[aeiou]` match any lowercase vowel character
- `[^aeiou]` match any character other than lowercase vowels

- `[a-d]` match any of `a` or `b` or `c` or `d`
  - the range metacharacter `-` can be applied between any two characters
- `\a` match any alphabet character `[a-zA-Z]`
- `\A` match other than alphabets `[^a-zA-Z]`
- `\l` match lowercase alphabets `[a-z]`
- `\L` match other than lowercase alphabets `[^a-z]`
- `\u` match uppercase alphabets `[A-Z]`
- `\U` match other than uppercase alphabets `[^A-Z]`
- `\d` match any digit character `[0-9]`
- `\D` match other than digits `[^0-9]`
- `\o` match any octal character `[0-7]`
- `\O` match other than octals `[^0-7]`
- `\x` match any hexadecimal character `[0-9a-fA-F]`
- `\X` match other than hexadecimals `[^0-9a-fA-F]`
- `\h` match alphabets and underscore `[a-zA-Z_]`
- `\H` match other than alphabets and underscore `[^a-zA-Z_]`
- `\w` match any word character (alphabets, digits, underscore) `[a-zA-Z0-9_]`
  - this definition is same as seen earlier with word boundaries
- `\W` match other than word characters `[^a-zA-Z0-9_]`
- `\s` match space and tab characters `[ \t]`
- `\S` match other than space and tab characters `[^ \t]`

Here are some examples with character classes:

- `c[ou]t` matches `cot` or `cut`
- `\<[ot][on]\>` matches `oo` or `on` or `to` or `tn` as whole words only
- `^[on]\{2,}$` matches `no` or `non` or `noon` or `on` etc as whole lines only
- `s/"[^"]\+"/X/g` replaces `"mango" and "(guava)"` with `X and X`
- `s/\d\+/-/g` replaces `Sample123string777numbers` with `Sample-string-numbers`
- `s/\<0*[1-9]\d\{2,}\>/X/g` replaces `0501 035 26 98234` with `X 035 26 X` (numbers >=100 with optional leading zeros)
- `s/\W\+/ /g` replaces `load2;err_msg--\ant` with `load2 err_msg ant`


To include the end-of-line character, use `\_` instead of `\` for any of the above escape sequences. For example, `\_s` will help you match across lines. Similarly, use `\_[]` for bracketed classes.


The above escape sequences do not have special meaning within bracketed classes. For example, `[\d\s]` will only match `\` or `d` or `s` . You can use named character sets in such scenarios. For example, `[[:digit:][:blank:]]` to match digits or space or tab characters. See :h :alnum: for full list and more details.


The predefined sets are also better in terms of performance compared to bracketed versions. And there are more such sets than the ones discussed above. See :h character-classes for more details.

Alternation and Grouping

Alternation helps you to match multiple terms and they can have their own anchors as well (since each alternative is a regexp pattern). Often, there are some common things among the regular expression alternatives. In such cases, you can group them using a pair of parentheses metacharacters. Similar to `a(b+c)d = abd+acd` in maths, you get `a(b|c)d = abd|acd` in regular expressions.

- `\|` match either of the specified patterns
  - `min\|max` matches `min` or `max`
  - `one\|two\|three` matches `one` or `two` or `three`
  - `\<par\>\|er$` matches the whole word `par` or a line ending with `er`
- `\(pattern\)` group a pattern to apply quantifiers, create a terser regexp by taking out common elements, etc
  - `a\(123\|456\)b` is equivalent to `a123b\|a456b`
  - `hand\(y\|ful\)` matches `handy` or `handful`
  - `hand\(y\|ful\)\?` matches `hand` or `handy` or `handful`
  - `\(to\)\+` matches `to` or `toto` or `tototo` and so on
  - `re\(leas\|ceiv\)\?ed` matches `reed` or `released` or `received`

There can be tricky situations when using alternation. Say, you want to match `are` or `spared` — which one should get precedence? The bigger word `spared` or the substring `are` inside it or based on something else? The alternative which matches earliest in the input gets precedence, irrespective of the order of the alternatives.

- `s/are\|spared/X/g` replaces `rare spared area` with `rX X Xa`
  - `s/spared\|are/X/g` will also give the same result

In case of matches starting from the same location, for example `spa` and `spared`, the left-most alternative gets precedence. Sort by longest term first if don't want shorter terms to take precedence.

- `s/spa\|spared/**/g` replaces `spared spare` with `**red **re`
- `s/spared\|spa/**/g` replaces `spared spare` with `** **re`

Backreference

The groupings seen in the previous section are also known as **capture groups**. The string captured by these groups can be referred later using a backreference `\N` where `N` is the capture group you want. Backreferences can be used in both search and replacement sections.

- `\(pattern\)` capture group for later use via backreferences
- `\%(pattern\)` non-capturing group
- leftmost group is `1`, second leftmost group is `2` and so on (maximum `9` groups)
- `\1` backreference to the first capture group
- `\2` backreference to the second capture group
- `\9` backreference to the ninth capture group
- `&` or `\0` backreference to the entire matched portion

Here are some examples:

- `\(\a\)\1` matches two consecutive repeated alphabets like `ee`, `TT`, `pp` and so on
  - recall that `\a` refers to `[a-zA-Z]`

- `\(\a\)\1\+` matches two or more consecutive repeated alphabets like `ee`, `ttttt`, `PPPPPPPP` and so on
- `s/\d\+/(&)/g` replaces `52 apples 31 mangoes` with `(52) apples (31) mangoes` (surround digits with parentheses)
- `s/\(\w\+\),\(\w\+\)/\2,\1/g` replaces `good,bad 42,24` with `bad,good 24,42` (swap words separated by comma)
- `s/\(_\)\?_/\1/g` replaces `_fig __123__ _bat_` with `fig _123_ bat` (reduce `__` to `_` and delete if it is a single `_`)
- `s/\(\d\+\)\%(abc\)\+\(\d\+\)/\2:\1/` replaces `12abcabcabc24` with `24:12` (match digits separated by one or more `abc` sequences, swap the numbers with `:` as the separator)
  - note the use of non-capturing group for `abc` since it isn't needed later
  - `s/\(\d\+\)\(abc\)\+\(\d\+\)/\3:\1/` does the same if only capturing groups are used

Referring to the text matched by a capture group with a quantifier will give only the last match, not the entire match. Use a capture group around the grouping and quantifier together to get the entire matching portion. In such cases, the inner grouping is an ideal candidate to use non-capturing group.

- `s/a \(\d\{3}\)\+/b (\1)/` replaces `a 123456789` with `b (789)`
  - `a 4839235` will be replaced with `b (923)5`
- `s/a \(\%(\d\{3}\)\+\)/b (\1)/` replaces `a 123456789` with `b (123456789)`
  - `a 4839235` will be replaced with `b (483923)5`

Lookarounds

Lookarounds help to create custom anchors and add conditions within the `searchpattern`. These assertions are also known as **zero-width patterns** because they add restrictions similar to anchors and are not part of the matched portions.

Vim's syntax is different than those usually found in programming languages like Perl, Python and JavaScript. The syntax starting with `\@` is always added as a suffix to the pattern atom used in the assertion. For example, `(?!\d)` and `(?<=pat.*)` in other languages are specified as `\d\@!` and `\(pat.*\)\@<=` respectively in Vim.

- `\@!` negative lookahead assertion
  - `ice\d\@!` matches `ice` as long as it is *not* immediately followed by a digit character, for example `ice` or `iced!` or `icet5` or `ice.123` but not `ice42` or `ice123`
  - `s/ice\d\@!/X/g` replaces `iceiceice2` with `XXice2`
  - `s/par\(.*\<par\>\)\@!/X/g` replaces `par` with `X` as long as whole word `par` is *not* present later in the line, for example `parse and par and sparse` is converted to `parse and X and sXse`
  - `at\(\(go\)\@!.\)*par` matches `cat,dog,parrot` but not `cat,god,parrot` (i.e. match `at` followed by `par` as long as `go` isn't present in between, this is an example of *negating a grouping*)
- `\@<!` negative lookbehind assertion
  - `_\@<!ice` matches `ice` as long as it is *not* immediately preceded by a `_` character, for example `ice` or `_(ice)` or `42ice` but not `_ice`

    - `\(cat.*\)\@<!dog` matches `dog` as long as `cat` is *not* present earlier in the line, for example `fox,parrot,dog,cat` but not `fox,cat,dog,parrot`
- `\@=` positive lookahead assertion
    - `ice\d\@=` matches `ice` as long as it is immediately followed by a digit character, for example `ice42` or `ice123` but not `ice` or `iced!` or `icet5` or `ice.123`
    - `s/ice\d\@=/X/g` replaces `ice ice_2 ice2 iced` with `ice ice_2 X2 iced`
- `\@<=` positive lookbehind assertion
    - `_\@<=ice` matches `ice` as long as it is immediately preceded by a `_` character, for example `_ice` or `(_ice)` but not `ice` or `_(ice)` or `42ice`


You can also specify the number of bytes to search for lookbehind patterns. This will significantly speed up the matching process. You have to specify the number between the `@` and `<` characters. For example, `_\@1<=ice` will lookback only one byte before `ice` for matching purposes. `\(cat.*\)\@10<!dog` will lookback only ten bytes before `dog` to check the given assertion.

Atomic Grouping

As discussed earlier, both greedy and non-greedy quantifiers will try to satisfy the overall pattern by varying the amount of characters matched by the quantifiers. You can use atomic grouping to safeguard a pattern from further backtracking. Similar to lookarounds, you need to use `\@>` as a suffix, for example `\(pattern\)\@>` .

- `s/\(0*\)\@>\d\{3,\}/(&)/g` replaces only numbers >= 100 irrespective of any number of leading zeros, for example `0501 035 154` is converted to `(0501) 035 (154)`
    - `\(0*\)\@>` matches the `0` character zero or more times, but it will not give up this portion to satisfy overall pattern
    - `s/0*\d\{3,\}/(&)/g` replaces `0501 035 154` with `(0501) (035) (154)` (here `035` is matched because `0*` will match zero times to satisfy the overall pattern)
- `s/\(::.\{-}::\)\@>par//` replaces `fig::1::spar::2::par::3` with `fig::1::spar::3`
    - `\(::.\{-}::\)\@>` will match only from `::` to the very next `::`
    - `s/::.\{-}::par//` replaces `fig::1::spar::2::par::3` with `fig::3` (matches from the first `::` to the first occurrence of `::par`)

Set start and end of the match

Some of the positive lookbehind and lookahead usage can be replaced with `\zs` and `\ze` respectively.

- `\zs` set the start of the match (portion before `\zs` won't be part of the match)
    - `s/\<\w\zs\w*\W*//g` replaces `sea eat car rat eel tea` with `secret`
    - same as `s/\(\<\w\)\@<=\w*\W*//g` or `s/\(\<\w\)\w*\W*/\1/g`
- `\ze` set the end of the match (portion after `\ze` won't be part of the match)
    - `s/ice\ze\d/X/g` replaces `ice ice_2 ice2 iced` with `ice ice_2 X2 iced`
    - same as `s/ice\d\@=/X/g` or `s/ice\(\d\)/X\1/g`


As per :h \zs and :h \ze, these "Can be used multiple times, the last one encountered in a matching branch is used."

Magic modifiers

These escape sequences change certain aspects of the syntax and behavior of the search pattern that comes after such a modifier. You can use multiple such modifiers as needed for particular sections of the pattern.

Magic and nomagic

- `\m` magic mode (this is the default setting)
- `\M` nomagic mode
  - `.` , `*` and `~` are no longer metacharacters (compared to magic mode)
  - `\.` , `\*` and `\~` will make them to behave as metacharacters
  - `^` and `$` would still behave as metacharacters
  - `\Ma.b` matches only `a.b`
  - `\Ma\.b` matches `a.b` as well as `a=b` or `a<b` or `acb` etc

Very magic

The default syntax of Vim regexp has only a few metacharacters like `.` , `*` , `^` and `$` . If you are familiar with regexp usage in programming languages such as Perl, Python and JavaScript, you can use `\v` to get a similar syntax in Vim. This will allow the use of more metacharacters such as `()` , `{}` , `+` , `?` and so on without having to prefix them with a `\` metacharacter. From :h magic documentation:

> Use of `\v` means that after it, all ASCII characters except `0-9` , `a-z` , `A-Z` and `_` have special meaning

- `\v<his>` matches `his` or `to-his` but not `this` or `history` or `_hist`
- `a<b.*\v<end>` matches `c=a<b #end` but not `c=a<b #bending`
  - note that `\v` is used after `a<b` to avoid having to escape the first `<`
- `\vone|two|three` matches `one` or `two` or `three`
- `\vabc+` matches `abc` or `abccc` but not `ab` or `bc`
- `s/\vabc?/X/` replaces `abcc` with `Xc`
- `s/\vt.{-}a/X/g` replaces `that is quite a fabricated tale` with `XX fabricaXle`
- `\vab{3}c` matches `xabbbcz` but not `abbc` or `abbbbbc`
- `s/\v(\w+),(\w+)/\2,\1/g` replaces `good,bad 42,24` with `bad,good 24,42`
  - compare this to the default mode: `s/\(\w\+\),\(\w\+\)/\2,\1/g`

Very nomagic

From :h magic documentation:

> Use of `\V` means that after it, only a backslash and terminating character (usually `/` or `?`) have special meaning

- `\V^.*{}$` matches `^.*{}$` literally
- `\V^.*{}$\.\*abcd` matches `^.*{}$` literally only if `abcd` is found later in the line
  - `\V^.*{}$\m.*abcd` can also be used
- `\V\^This` matches `This is a sample` but not `Do This`
- `\V)\$` matches `apple (5)` but not `def greeting():`

Case sensitivity

These will override flags and settings, if any. Unlike the magic modifiers, you cannot apply `\c` or `\C` for a specific portion of the pattern.

- `\c` case insensitive search
  - `\cthis` matches `this` or `This` or `THIs` and so on
    - `th\cis` or `this\c` and so on will also result in the same behavior
- `\C` case sensitive search
  - `\Cthis` match exactly `this` but not `This` or `THIs` and so on
    - `th\Cis` or `this\C` and so on will also result in the same behavior
- `s/\Ccat/dog/gi` replaces `cat Cat CAT` with `dog Cat CAT` since the `i` flag gets overridden

Changing Case

These can be used in the replacement section:

- `\u` Uppercases the next character
- `\U` UPPERCASES the following characters
- `\l` lowercases the next character
- `\L` lowercases the following characters
- `\e` or `\E` will end further case changes
- `\L` or `\U` will also override any existing conversion

Examples:

- `s/\<\l/\u&/g` replaces `hello. how are you?` with `Hello. How Are You?`
  - recall that `\l` in the search section is equivalent to `[a-z]`
- `s/\<\L/\l&/g` replaces `HELLO. HOW ARE YOU?` with `hELLO. hOW aRE yOU?`
  - recall that `\L` in the search section is equivalent to `[A-Z]`
- `s/\v(\l)_(\l)/\1\u\2/g` replaces `aug_price next_line` with `augPrice nextLine`
- `s/.*/\L&/` replaces `HaVE a nICe dAy` with `have a nice day`
- `s/\a\+/\u\L&/g` replaces `HeLLo:bYe gOoD:beTTEr` with `Hello:Bye Good:Better`
  - `s/\a\+/\L\u&/g` can also be used in this case
- `s/\v(\a+)(:\a+)/\L\1\U\2/g` replaces `Hi:bYe gOoD:baD` with `hi:BYE good:BAD`

Alternate delimiters

From :h substitute documentation:

> Instead of the `/` which surrounds the pattern and replacement string, you can use any other single-byte character, but not an alphanumeric character, `\` , `"` or `|` . This is useful if you want to include a `/` in the search pattern or replacement string.

- `s#/home/learnbyexample/#\~/#` replaces `/home/learnbyexample/reports` with `~/reports`
    - compare this with `s/\/home\/learnbyexample\//\~\//`

Escape sequences

Certain characters like tab, carriage return, newline, etc have escape sequences to represent them. Additionally, any character can be represented using their codepoint value in decimal, octal and hexadecimal formats. Unlike character set escape sequences like `\w`, these can be used inside character classes as well. If the escape sequences behave differently in `searchpattern` and `replacestring` portions, they'll be highlighted in the descriptions below.

- `\t` tab character
- `\b` backspace character
- `\r` matches carriage return for `searchpattern`, produces newline for `replacestring`
- `\n` matches end-of-line for `searchpattern`, produces ASCII NUL for `replacestring`
    - `\n` can also match `\r` or `\r\n` (where `\r` is carriage return) depending upon the `fileformat` setting
- `\%d` matches character specified by decimal digits
    - `\%d39` matches the single quote character
- `\%o` matches character specified by octal digits
    - `\%o47` matches the single quote character
- `\%x` matches character specified by hexadecimal digits (max 2 digits)
    - `\%x27` matches the single quote character
- `\%u` matches character specified by hexadecimal digits (max 4 digits)
- `\%U` matches character specified by hexadecimal digits (max 8 digits)

Using `\%` sequences to insert characters in `replacestring` hasn't been implemented yet. See vi.stackexchange: Replace with hex character for workarounds.

See ASCII code table for a handy cheatsheet with all the ASCII characters and conversion tables. See codepoints for Unicode characters.

Escaping metacharacters

To match the metacharacters literally (including character class metacharacters like `-`), i.e. to remove their special meaning, prefix those characters with a `\` (backslash) character. To indicate a literal `\` character, use `\\`. Depending on the pattern, you can also use a different magic modifier to reduce the need for escaping. Assume default magicness for the below examples unless otherwise specified.

- `^` and `$` do not require escaping if they are used out of position
    - `b^2` matches `a^2 + b^2 - C*3`
    - `$4` matches `this ebook is priced $40`
    - `\^super` matches `^superscript` (you need the `\` here since `^` is at the customary position)
- `[` and `]` do not require escaping if only one of them is used
    - `b[1` matches `ab[123`

    - `42]` matches `xyz42] =`
    - `b\[123]` or `b[123\]` matches `ab[123] = d`
- `[` in the substitute command requires careful consideration
    - `s/b[1/X/` replaces `b[1/X/` with nothing
    - `s/b\[1/X/` replaces `ab[123` with `aX23`
- `\Va*b.c` or `a\*b\.c` matches `a*b.c`
- `&` in the replacement section requires escaping to represent it literally
    - `s/and/\&/` replaces `apple and mango` with `apple & mango`

The following can be used to match character class metacharacters literally in addition to escaping them with a `\` character:

- `-` can be specified at the start or end of the list, for example `[-0-5]` and `[a-z-]`
- `^` should be other than the first character, for example `[+a^.]`
- `]` should be the first character, for example `[]a-z]` and `[^]a]`

Replacement expressions

- `\=` when `replacestring` starts with `\=` , it is treated as an expression
- `s/date:\zs/\=strftime("%Y-%m-%d")/` appends the current date
    - for example, changes `date:` to `date:2024-06-25`
- `s/\d\+/\=submatch(0)*2/g` multiplies matching numbers by 2
    - for example, changes `4 and 10` to `8 and 20`
    - `submatch()` function is similar to backreferences, `0` gives the entire matched string, `1` refers to the first capture group and so on
- `s/\(.*\)\zs/\=" = " . eval(submatch(1))/` appends result of an expression
    - for example, changes `10 * 2 - 3` to `10 * 2 - 3 = 17`
    - `.` is the string concatenation operator
    - `eval()` here executes the contents of the first capture group as an expression
- `s/"[^"]\+"/\=substitute(submatch(0), '[aeiou]', '\u&', 'g')/g` affects vowels only inside double quotes
    - for example, changes `"mango" and "guava"` to `"mAngO" and "gUAvA"`
    - `substitute()` function works similarly to the `s` command
    - first argument is the text to work on
    - second argument is similar to `searchpattern`
    - third argument is similar to `replacestring`
    - fourth argument is flags, use an empty string if not required
    - see :h substitute() for more details and differences compared to the `s` command
- `perldo s/\d+/$&*2/ge` changes `4 and 10` to `8 and 20`
    - useful if you are familiar with Perl regular expressions and the `perl` interface is available with your Vim installation
    - note that the default range is `1,$` (the `s` command works only on the current line by default)
    - see :h perldo for restrictions and more details


See :h usr_41.txt for details about Vim script.


See :h sub-replace-expression for more details.

 See also stackoverflow: find all occurrences and replace with user input.

Miscellaneous

- `\%V` match inside the visual area only
  - `s/\%V10/20/g` replaces `10` with `20` only inside the visual area
  - without `\%V` , the replacement would happen anywhere on the lines covered by the visual selection
- `\%[set]` match zero or more of these characters in the same order, as much as possible
  - `spa\%[red]` matches `spa` or `spar` or `spare` or `spared` (longest match wins)
    - ⋆ same as `\vspa(red|re|r)?` or `\vspa(red?|r)?` and so on
  - `ap\%[[pt]ly]` matches `ap` or `app` or `appl` or `apply` or `apt` or `aptl` or `aptly`
- `\_^` and `\_$` restrict the match to start-of-line and end-of-line respectively, useful for multiline patterns
- `\%^` and `\%$` restrict the match to start-of-file and end-of-file respectively
- `~` represents the last replacement string
  - `s/apple/banana/` followed by `/~` will search for `banana`
  - `s/apple/banana/` followed by `s/fig/(~)/` will use `(banana)` as the replacement string

Further Reading

- vi.stackexchange: How to find and replace in Vim without having to type the original word? — lots of tips and tricks
- vi.stackexchange: How to replace each match with incrementing counter?
- vi.stackexchange: What is the rationale for \r and \n meaning different things in s command? and stackoverflow: Why is \r a newline for Vim?
- stackoverflow: What does this regex mean?

Macro

The `.` repeat command repeats only the last change. And it gets overwritten with every change. The `q` command allows you to record a *sequence of commands* and execute them later whenever you need. You can make it recursive, add a count prefix, combine it with Command-line mode commands and so on. Powerful indeed!

With so many built-in features, sometimes it isn't easy to choose. I prefer the substitute command to macros if both of them can be used for the given problem, especially if the processing doesn't require multiple lines to be considered at once for the solution. That said, macros are more flexible, having an inherent advantage of being able to easily integrate numerous Vim commands. Also, macros allow you to progress in smaller chunks, which might be easier compared to a complicated regexp based solution.

Documentation links:

- :h 10.1 — record and playback commands
- :h complex-repeat — reference manual for `q` and related commands

Macro usage steps

Here's a rough overview of the `q` command usage. Working examples will be discussed in later sections.

1) Press `q` to start the recording
2) Use any alphanumeric character as the register to store the recording (for example, `a`)
3) Execute command sequence to accomplish the required task
4) Press `q` again to stop the recording
5) Press `@a` (the register used in step 2) to execute the recorded command sequence
    - `5@a` execute the macro `5` times
    - `@@` repeat the last executed macro


Command-line area will show `recording @a` after step 2 and this indicator vanishes after step 4.


Note that these registers are shared across recording, delete and yank commands. You'll see how this helps to modify a recording later, but you should also be careful not to mix them if you want separate recording and paste use cases. As mentioned earlier in the Normal mode chapter, uppercase registers will append to existing content in lowercase registers.


See also vi.stackexchange: Can I repeat a macro with the "dot operator"? (one of the solutions will allow you to use the `.` command to execute a macro immediately after recording as well).

Example 1

The `qwceHello^[q` macro recording clears text till the end of the word and inserts `Hello`. Here's a breakdown of this command sequence:

- `q` start recording
- `w` register used to save the macro
- `ce` change till the end of the word
- `Hello` insert these characters
- `^[` this is a single character that denotes the `Esc` key
  - in other words, press `Esc` key for this step, *don't* type the `^` and `[` characters
  - you'll see this representation if you paste the contents of the `"w` register using `"wp`
- `q` stop recording

After you've recorded the macro, you can execute this command sequence anywhere else you need it. For example, if the cursor is on the fourth character of the text `Hi-there` and `@w` is pressed, you'll get `Hi-Hello`.

Modifying a macro

As mentioned earlier, registers are shared across recording, delete and yank commands. When you call a macro using `@`, the register content is treated as the sequence of commands to be executed. So, editing a register's content will automatically update the behavior of the macro as well. Knowing that you can modify a macro also helps if you make a mistake — you can choose to finish the recording and update later instead of restarting the recording.

Suppose you want to use `'Hello!'` instead of `Hello` for the macro discussed in the previous section. Here's one possible way to make the changes:

- `"wp` paste the contents of `"w` register
  - you should get `ceHello^[`
- `ce'Hello'!^[` modify the text as needed
- `"wy` update the contents of `"w` register after visually selecting the modified text or using motion commands in Normal mode

After you've modified the register contents, check if it is working as expected. For example, if the cursor is on the fourth character of the text `Hi-there` and `@w` is pressed, you should now get `Hi-'Hello'!`.

In case you wish to create a new macro from scratch by just typing the required text instead of using the `q` command, you'll find `Ctrl + v` (or the `Ctrl + q` alias) useful to insert keys like `Esc` and `Enter`. To do so, press `Ctrl + v` followed by the required key. You'll get `^[` for `Esc`, `^M` for `Enter` and so on.

`let @w = "ce'Hello'!^["` adding this line to the `vimrc` file will load the `"w` register with the given text at startup.

Example 2

Suppose you forgot to add curly braces for single statement control structures in a Perl program:

```
# syntax error
if($word eq reverse $word)
    print "$word is a palindrome\n";

# corrected code
if($word eq reverse $word)
{
    print "$word is a palindrome\n";
}
```

`qpo{^[jo}^[q` is one way to do it:

- `qp` start recording and use register `p`
- `o` open a new line
- `{` insert the `{` character
- `^[` go back to Normal mode (`Esc` key)
- `j` move down one line
- `o` open another line
- `}` insert the `}` character
- `^[` go back to Normal mode (`Esc` key)
- `q` stop recording

Having a macro will help you apply this correction whenever you forget braces for single statement control structures.

 Note that `{` and `}` will be indented based on style settings for that particular filetype.

Example 3

I used `F`r[f`s]()^["*P` macro to replace `` `:h <topic>` `` with a hyperlink to the corresponding online help page for this ebook. Assume the cursor is somewhere within the `:h <topic>` text portion surrounded by backticks (markdown formatting for inline code). This has to be changed to `[:h <topic>](link)` (markdown formatting for hyperlinks).

- `F`` move cursor to the starting backtick
- `r[` replace backtick with `[`
- `f`` move cursor to the ending backtick
- `s]()` replace backtick with `]()`
- `^[` go back to Normal mode (`Esc` key)
- `"*P` paste contents of the last highlighted text selection
    - note the use of uppercase `P` to paste content to the left of the cursor

Once the macro was recorded, I just had to select the url from the browser for each help topic and execute the macro. I used `n` to navigate in the markdown files after using `:h` as the search pattern.

Motion and Filter

If you have to apply the same macro for text portions that are next to each other, you can add motion commands at the end of the macro for reaching the next text portion. The motion command could be arrow motions, searching using `/` and so on. Doing so will allow you to use a count prefix to apply the macro for all the text portions in one shot. This assumes that you can easily count the number of text portions. For example, consider this Python snippet where you want to change single line definitions to multiple lines:

```
def square(n): return n ** 2
def cube(n): return n ** 3
def isodd(n): return n % 2 == 1
```

You can do a recording as usual, select these lines visually (or use a range) and then apply the macro using `normal @d` in Command-line mode. Or, you could add a motion to automatically go to the next line and use a count prefix as described below.

`qd0f:lr^M>>o^[jq` is one way to achieve this:

- `0f:l` Move to beginning of the line and then move the cursor to the character after the first occurrence of `:` (which is a space character in the above snippet)
    - this also assumes that there won't be any `:` character as part of the function arguments
- `r^M` replace the space character with a newline character
- `>>` indent the line
    - note that this won't be required if indentation is automatically applied based on Python syntax
- `o^[` open a new line and go back to Normal mode
- `j` move to the next line (this makes it possible to use the count prefix)

After recording, you can use `3@d` on the first line to get the output as shown below:

```
def square(n):
    return n ** 2

def cube(n):
    return n ** 3

def isodd(n):
    return n % 2 == 1
```

Suppose the Python function definitions discussed above aren't next to each other but can be found anywhere in the Python script file. In such cases, if you are able to reliably identify the lines using a regexp filter, you can use the `:g` command.

- `qdf:lr^M>>o^[q` simplified macro
    - `0` not required since the cursor starts at the beginning
    - no need to move to the next line
- `:g/^def .*): / normal @d` apply the macro for filtered lines
- `:%s/^def .*):\zs \(.*\)/\r\t\1\r/` if you are comfortable with regexp, you could also just use the substitution command like this one instead

Recursive recording

Suppose it isn't easy to count the number of text portions and filtering is complicated too. In such cases, you might be able to use recursive recording that continues to execute the macro until one of the steps fails. Similar to recursive function calls, you have to call the macro from within the recording. Consider this Python snippet where you want to change single line definitions to multiple lines:

```
def square(n): return n ** 2
def cube(n): return n ** 3
def isodd(n): return n % 2 == 1
print(square(12))
```

`qr0f:lr^M>>o^[j@rq` is one way to achieve this. The only addition here is `@r` at the end of the recording compared to the solution discussed in the previous section. For the fourth line with `print()` function, the macro will stop when it doesn't find the `:` character. It would've stopped even if a `:` was found, provided it was the last character, since the `l` motion would've failed.

Using `@r` on the first line of the above snippet would give the following output:

```
def square(n):
    return n ** 2

def cube(n):
    return n ** 3

def isodd(n):
    return n % 2 == 1

print(square(12))
```


Note that the register being used here must be empty before you start the recording, otherwise you might see some unwanted changes when you type `@r` while recording. To ensure this register is empty, you can use `qrq` (i.e. record an empty macro) before you record the recursive macro.


If the `:s` command is part of the recording and you do not want the macro to stop if the search pattern isn't found, you can use the `e` flag.

Here are some more examples:

- vi.stackexchange: How do I stop a recursive macro at the end of the line? — one of the examples shows how to incorporate Vimscript, so you get full programming capabilities like variables, `if` control structure and so on
- vi.stackexchange: How to reverse every 4 lines?
- vi.stackexchange: Correct all spelling mistakes in the document

Exercise

Given the following text:

```
# Introduction
# Normal mode
# Command Line mode
# Visual mode
```

Use a macro (or the substitute command if you prefer) to get the modified text as shown below:

```
* [Introduction](#introduction)
* [Normal mode](#normal-mode)
* [Command Line mode](#command-line-mode)
* [Visual mode](#visual-mode)
```

Further Reading

- Advanced vim macros
- Vim Macro Trickz
- vi.stackexchange: top Q&A on macro

Customizing Vim

Settings like indentation and keyword-pairs can vary between different programming languages and file types. You might need to adapt style guides based on client requirements. Or perhaps, you wish to create or override commands to suit your preferences.

This chapter will discuss how you can customize Vim for different purposes. Some of the settings will be specific to GVim.

Documentation links:

- :h usr_05.txt — set your settings
- :h usr_40.txt — make new commands
- :h usr_41.txt — write a Vim script
- :h usr_43.txt — using filetypes
- :h options.txt — reference manual for Options
- :h map.txt — reference manual for Key mapping, abbreviations and user-defined commands
- :h autocmd.txt — reference manual for Automatic commands

Editing vimrc

From :h usr_41.txt and :h vimrc-intro:

> The Vim script language is used for the startup vimrc file, syntax files, and many other things.
> The vimrc file can contain all the commands that you type after a colon. The simplest ones are for setting options.

This chapter only covers some use cases. You'll see what some of the settings do, how to use mappings, abbreviations and so on. Not much will be discussed about the programming aspects of Vim script. Make sure you have a `vimrc` file using the following details:

- `:e $MYVIMRC` if you already have a `vimrc` file, you can use this predefined variable to open it
- :h vimrc to find out where the `vimrc` file should be located for your OS
- `:source $MYVIMRC` apply changes from within your current Vim session

To view a sample `vimrc` file, I have one on GitHub. More resources are mentioned in the **Further Reading** section at the end of this chapter.

defaults.vim

If you haven't created a `vimrc` file, the `defaults.vim` file that comes with Vim installation will be used. This file aims to provide saner defaults like enabling syntax highlighting, filetype settings and so on.

- `source $VIMRUNTIME/defaults.vim` add this to your `vimrc` file if you want to keep these defaults
- :h defaults.vim-explained describes the settings provided by `defaults.vim`

Alternatively, you can copy only the parts you want to retain from the `defaults.vim` file to your `vimrc` file.

General Settings

 `set` syntax and guidelines were introduced in the Setting options section.

- `set history=200` increase default history from 50 to 200
  - as mentioned in the Command-line mode chapter, there are separate history lists for `:` commands, search patterns, etc
- `set nobackup` disable backup files
- `set noswapfile` disable swap files
- `colorscheme murphy` a dark theme
  - you can use `:colorscheme` followed by a space and then press `Tab` or `Ctrl` + `d` to get a list of the available color schemes
- `set showcmd` Command-line area will show partial Normal mode commands and character/line/block-selection for Visual mode
- `set wildmode=longest,list,full` use `bash` -like tab completion
  - first tab will complete as much as possible
  - second tab will provide a list
  - third and subsequent tabs will cycle through the completion options

 `:h 'history'` will give you the documentation for the given option (note the use of single quotes).

 You can use these settings from the Command-line mode as well, but will be active for the current Vim session only. Settings specified in the `vimrc` file will be loaded automatically at startup. You can also load a different file as the `vimrc`, which will be discussed in the CLI options chapter.

Further Reading

- stackoverflow: Vim backup files
- stackoverflow: Disabling swap files
- stackoverflow: How to set persistent Undo

Text and Indent Settings

- `filetype plugin indent on` enables loading of `plugin` and `indent` files
  - these files become active based on the type of the file to influence syntax highlighting, indentation, etc
  - `:echo $VIMRUNTIME` gives your installation directory (`indent` and `plugin` directories would be present in this path)
  - see :h vimrc-filetype, :h :filetype-overview and :h filetype.txt for more details
- `set autoindent` copy indent from the current line when starting a new line
  - useful for files not affected by the `indent` setting
  - see also :h smartindent
- `set textwidth=80` guideline for Vim to automatically move to a new line with `80` characters as the limit

    - white space is used to break lines, so a line can still be greater than the limit if there's no white space
    - default is `0` which disables this setting
- `set colorcolumn=80` create a highlighted vertical bar at column number `80`
    - use `highlight ColorColumn` setting to customize the color for this vertical bar
    - see vi.stackexchange: Keeping lines to less than 80 characters for more details
- `set shiftwidth=4` number of spaces to use for indentation (default is `8`)
- `set tabstop=4` width for the tab character (default is `8`)
- `set expandtab` use spaces for tab expansion
- `set cursorline` highlight the line containing the cursor

Search Settings

- `set hlsearch` highlight all matching portions
    - using `:noh` (short for `:nohlsearch`) will clear the currently highlighted portions
- `set incsearch` highlights current match as you type the pattern, the screen is updated automatically as needed
    - pressing the `Enter` key would move the cursor to the matched portion
    - pressing the `Esc` key would keep the cursor at the current location
    - other matching terms will be highlighted based on the `hlsearch` setting

Custom mapping

Mapping helps you to create new commands or redefine existing ones. You can restrict such mappings for specific modes as well. Only the following settings will be discussed in this chapter:

- `nnoremap` Normal mode non-nested, non-recursive mapping
- `xnoremap` Visual mode non-nested, non-recursive mapping
- `inoremap` Insert mode non-nested, non-recursive mapping
- `inoreabbrev` Insert mode non-nested, non-recursive abbreviation

The following will not be discussed, but you might find it useful to know or explore further:

- `nmap` , `xmap` , `imap` and `iabbrev` allows nested and recursive mappings
- `nunmap` , `xunmap` , `iunmap` and `iunabbrev` unmaps the given command (usually used from Command-line mode to temporarily disable a mapping, will be available again on startup if it was defined in `vimrc`)
    - use `mapclear` instead of `unmap` to clear all the mappings for that particular mode
- `onoremap` (or `omap`) map a motion or text object to be used with commands like `d` or `y`
- `command` helps you create a Command-line mode command, see :h 40.2 and :h user-commands for details

 `:nmap` , `:xmap` , `:imap` and `:iab` will list all the current mappings for that particular mode. You can provide an argument to display the mapping for that particular command, for example `:nmap Y` . See :h key-mapping and :h map-overview for reference manuals.

Normal mode

- `nnoremap <F2> :w<CR>` press `F2` function key to save changes
  - `<F2>` represents the `F2` function key and `<CR>` represents the `Enter` key
  - I chose `F2` since it is close to the `Esc` key (`F1` opens help page)
- `nnoremap <F3> :wq<CR>` press `F3` to save changes and quit
- `nnoremap <F4> ggdG` press `F4` to delete everything
- `nnoremap <F5> :%y+<CR>` press `F5` to copy everything to the system clipboard
- `nnoremap <left> <nop>` do nothing when the `←` arrow key is pressed
  - likewise, you can map the other arrow keys to do nothing
- `nnoremap Y y$` change `Y` to behave similarly to `D` and `C`
- `nnoremap / /\v` add very magic mode modifier for forward direction search
- `nnoremap ? ?\v` add very magic mode modifier for backward direction search
- `nnoremap <silent> <Space> :noh<CR><Space>` press `Space` key to clear the currently highlighted portions
  - `<silent>` modifier executes the command without displaying in the Command-line area
  - Note that this mapping also retains the default behavior of the `Space` key
- `nnoremap <A-1> 1gt` press `Alt` + `1` to switch to the first tab
  - I prefer this to make switching tabs consistent with browser and terminal shortcuts
- `nnoremap <A-2> 2gt` press `Alt` + `2` to switch to the second tab and so on

 See :h map-which-keys to know which keys are not already Vim commands, which ones are not commonly used, etc.

 See :h key-notation for a list of keys that can be represented using the `<>` notation.

Map leader

Normal mode commands are already crowded, so if you are looking to create new commands, using a leader mapping can help you out. You can define a key that'll serve as a prefix for these new set of commands. By default, the backslash key is used as the leader key.

- `nnoremap <Leader>f gg=G` if `mapleader` hasn't been set, using `\f` will auto indent the code for the whole file
- `let mapleader = ";"` change the leader key to `;`
  - `nnoremap <Leader>f gg=G` this will now require `;f` since the leader key was changed

 See learnvimscriptthehardway: Leaders for more examples and details.

Insert mode

- `inoremap <F2> <C-o>:w<CR>` press `F2` to save changes in Insert mode as well
  - `Ctrl` + `o` is used here to execute a command and return back to Insert mode automatically

    - `imap <F2> <C-o><F2>` can also be used if you've already defined the Normal mode mapping
- `inoremap <C-f> <Esc>ea` press `Ctrl` + `f` to move to the end of the word
    - I'd prefer `Ctrl` + `e` but that is useful to cancel autocompletion
- `inoremap <C-b> <C-Left>` press `Ctrl` + `b` to move to the beginning of the word
- `inoremap <C-a> <End>` press `Ctrl` + `a` to move to the end of the line
- `inoremap <C-s> <Home>` press `Ctrl` + `s` to move to the start of the line
- `inoremap <C-v> <C-o>"+p` press `Ctrl` + `v` to paste from the clipboard
    - If you need `Ctrl` + `v` functionality, the `Ctrl` + `q` alias can be used to insert characters like `Enter` key (but this alias may not work in some terminals)
- `inoremap <C-l> <C-x><C-l>` press `Ctrl` + `l` to autocomplete matching lines
    - See :h i_CTRL-x and :h ins-completion for all the features offered by `Ctrl` + `x`

 Use `noremap!` if you want a mapping to work in both Insert and Command-line modes.

Visual mode

- `xnoremap * y/<C-R>"<CR>` press `*` to search the visually selected text in the forward direction
    - recall that `Ctrl` + `r` helps you insert register contents in Command-line mode
- `xnoremap # y?<C-R>"<CR>` press `#` to search the visually selected text in the backward direction

 Note that `xnoremap` is used here since `vnoremap` affects both Visual and Select modes.

Abbreviations

Abbreviations are usually used to correct typos and insert frequently used text. From :h abbreviations documentation:

> 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.

- `inoreabbrev p #!/usr/bin/env perl<CR>use strict;<CR>use warnings;<CR>` expand `p` to the text as shown in the code snippet below
    - you can trigger the abbreviation completion using non-keyword characters such as `Esc`, `Space` and `Enter` keys, punctuation characters and so on
    - use `Ctrl` + `]` to expand the abbreviation without adding anything extra

```
#!/usr/bin/env perl
use strict;
```

```
use warnings;
```

- `inoreabbrev py #!/usr/bin/env python3` expand `py` to `#!/usr/bin/env python3`
  - this might cause issues if you need `py` literally (for example, `script.py`)
  - you can use something like `[p` or `@p` instead
- `inoreabbrev teh the` automatically correct `teh` typo to `the`
- `inoreabbrev @a always @()<CR>begin<CR>end<Esc>2k$` expand `@a` to the text as shown in the code snippet below
  - this one works best when you type `@a` followed by the `Esc` key to place the cursor at the end of the first line

```
always @()
begin
end
```

- `:abbreviate` or `:ab` list all abbreviations

 See :h 24.7 for more details about using abbreviations.

Matching Pairs

- `set matchpairs+=<:>` add `<>` to the list of pairs matched by the `%` command in Normal mode

 To match keywords like `if` - `else` pairs with `%` , you can use the `matchit.vim` plugin. This supports filetypes such as HTML, Vim, LaTeX, XML, etc. See :h matchit-install for more details.

GUI options

- `set guioptions-=m` remove menu bar
- `set guioptions-=T` remove tool bar

 See :h guioptions for more details.

Third-party customizations

 See :h 'runtimepath' to know the path within which you can add the plugins and packages discussed in this section. `~/.vim` is commonly used on Unix/Linux systems.

Make sure to backup your directory (`~/.vim` for example) and the `vimrc` file, so that you can easily apply your customizations on a new machine.

plugin

Some plugins are loaded by default. Some come with Vim installation but you have to explicitly enable them. You can also write your own or add plugins written by others. From :h add-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.
> There are two types of plugins:
> - global plugin: Used for all kinds of files
> - filetype plugin: Only used for a specific type of file

If you want to add a global plugin created by you or someone else, place it in the `plugin` directory. If you don't have that directory yet, you can create it using the below command (assuming Unix/Linux):

```
$ mkdir -p ~/.vim/plugin
$ cp plugin_file.vim ~/.vim/plugin/
```

If you have multiple related plugin files, you can put them under a subdirectory:

```
$ mkdir -p ~/.vim/plugin/python
$ cp file_1.vim file_2.vim ~/.vim/plugin/python/
```

If you want to add plugins that should work based on a specific filetype, add them to the `ftplugin` directory:

```
$ mkdir -p ~/.vim/ftplugin
$ cp ftplugin_file.vim ~/.vim/ftplugin/
```

package

Packages make it easy to manage projects that require multiple plugins, use a version controlled repository directly and so on. See :h packages for more details. From :h add-package:

> 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.
> The Vim distribution comes with a few packages that you can optionally use. For example, the matchit plugin.

- `packadd! matchit` enable `matchit` package
  - this plugin comes with Vim, see :h matchit for further details
  - `!` is used to prevent loading this plugin when Vim is started with the `--noplugin` CLI option

vim-surround is used here as an example for a third-party package. Installation instructions (provided in this repository) are shown below, assuming you want to enable this package at startup:

```
# 'pack' is the directory for packages
# 'tpope' subdirectory is useful to group all packages by this author
# 'start' means this package will be loaded at startup
$ mkdir -p ~/.vim/pack/tpope/start
```

```
# go to the directory and clone the git repository
# you can then update the repository when new changes are needed
$ cd ~/.vim/pack/tpope/start
$ git clone https://github.com/tpope/vim-surround.git
```

When you start Vim after the above steps, `vim-surround` will be automatically active. Couple of examples are shown below, see the repository linked above for more details.

- `ysiw]` will surround a word with `[]` , for example `hello` to `[hello]`
- `cs"'` will change text surrounded by double quotes to single quotes, for example `"hi bye"` to `'hi bye'`

If you want to enable this package optionally, put it under `opt` directory instead of `start` .

```
# 'opt' makes it optional
$ mkdir -p ~/.vim/pack/tpope/opt
$ cd ~/.vim/pack/tpope/opt
$ git clone https://github.com/tpope/vim-surround.git
```

- `:packadd vim-surround` enable this package from Command-line mode
- `packadd! vim-surround` enable this package in `vimrc` (usually under some condition)

color scheme

There are different ways to add a new color scheme. The simplest is to copy the `theme.vim` file to the `~/.vim/colors` directory. Or, follow the installation steps provided by the theme creators. Here are couple of solarized themes you can check out:

- vim-solarized
- vim-solarized8

After installation, you can use the `:colorscheme` command to set the new theme. If the theme offers multiple variations, you might need additional settings like `set background=dark` or `set background=light` . See the installation instructions provided in the above repositories for more details.

See **Where to put what** section under :h packages for more details about installation directories.

 See also this collection of awesome color schemes for Vim.

autocmd

From :h 40.3:

> 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.
> 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.

Syntax from the reference manual is shown below:

```
:au[tocmd] [group] {event} {aupat} [++once] [++nested] {cmd}
```

Here's an example for Python files:

```
augroup pyg
    autocmd!

    " add Python shebang for a new buffer with .py extension
    " py abbreviation was discussed earlier in this chapter
    autocmd BufNewFile *.py normal ipy

    " Black command is provided by a Python code formatter plugin
    autocmd BufWritePre *.py Black
augroup END
```

- `autocmd BufNewFile *.py normal ipy`
  - `BufNewFile` event that triggers on editing a file that doesn't already exist
  - `*.py` filenames ending with `.py` (similar to shell wildcards)
  - `normal ipy` command to be executed (`normal` is needed here since by default commands are treated as Command-line mode)
- `autocmd BufWritePre *.py Black`
  - `BufWritePre` event that triggers on writing a file
  - `Black` command to be executed (see black vim plugin documentation for more details)
- `augroup` helps you to group related autocommands
- `autocmd!` removes all autocommands within a group (`pyg` in the above example)
  - useful to avoid autocommands getting defined again when you source the `vimrc` file
- `:autocmd` list all autocommands, you can provide arguments to narrow down this listing

 Note that in earlier versions of Vim, double quotes is used for comments as shown in the above snippet. You'll need to use the `#` character instead for `vim9script` . See vim9-conversion-aid for upgrading old scripts.

See also:

- :h 40.3 for user manual, :h :autocmd and :h autocmd-groups for reference manuals
- :h autocmd-events for a list of events
- learnvimscriptthehardway: autocmd tutorial
- learnvimscriptthehardway: augroup tutorial

Further Reading

- Learn Vimscript the Hard Way — book on Vimscript and customizing Vim (written for `version 7.3`)
- Vimscript cheatsheet
- Vim Awesome — a directory of Vim plugins
- `vimrc` reference, tips and generation
  - stackoverflow: useful vimrc tips
  - vi.stackexchange: How do I debug my vimrc file?
  - vim-sensible

    - minimal vimrc for new users
    - Vim Configuration From Scratch
    - vimconfig — generate `vimrc` by selecting options
- vi.stackexchange: Open filename under cursor in a new tab (or split)
- stackoverflow: Open filename under cursor based on current filetype
- stackoverflow: Information regarding Vim history
- stackoverflow: Indenting all the files in a folder

CLI options

This chapter discusses some of the options you can use when starting Vim from the command line. A Unix/Linux distribution is assumed for the examples shown in this chapter. Syntax and features might vary for other platforms like Windows.

Documentation links:

- :h vim-arguments — reference manual for Vim arguments


Recall that you need to add `-` prefix for built-in help on CLI options, :h -y for example.

Default

- `gvim` opens a new unnamed buffer when a filename is not specified
- `gvim script.py` opens `script.py`
  - creates a blank buffer if `script.py` doesn't exist, file will be created only after you explicitly issue write commands
- `gvim report.log power.log area.log` opens the specified files
  - first file (`report.log` here) will be the current buffer
- `gvim -- *.txt` if filenames can start with `-` , use `--` to prevent such files from being treated as an option

Help

- `gvim -h` brief description of the options
  - not all options are discussed in this chapter, so you can use this to view the full list

Tabs and Splits

- `gvim -p *.log` opens the specified files as separate tab pages
  - by default, you can open a maximum of `10` pages, use the `tabpagemax` setting if you want to change this number
- `gvim -o *.log` opens the specified files as horizontal splits
- `gvim -O *.log` opens the specified files as vertical splits


You can append a number to each of these options to specify how many tabs or splits you want. For example, `gvim -p3 *.py` opens three tabs irrespective of the number of input files. Empty buffers will be used if there aren't enough input files to satisfy the given number.

Easy mode

- `gvim -y` opens in Insert mode and behaves like a click-and-type editor
  - useful for those who just want a simple text editor
  - or, perhaps you can prank Vim users by setting `alias vim='vim -y'`
  - use `Ctrl` + `l` or `Ctrl` + `o` if you want to use Normal mode commands

 See also novim-mode plugin, which aims to make Vim behave more like a *normal* editor.

Readonly and Restricted modes

- `gvim -R` Readonly mode
  - changes can still be made and saved despite warning messages
  - for example, by using `:w!`
- `gvim -M` *stricter* Readonly mode
  - changes cannot be made unless `:set modifiable` is used
  - file cannot be saved until `:set write` is used
- `gvim -Z` Restricted mode
  - commands using external shell are not allowed
  - for example, you won't be able to use `:!ls`

Cursor position

- `gvim + script.py` opens `script.py` and the cursor is placed on the last line
- `gvim +25 script.py` opens `script.py` and the cursor is placed on the 25th line
  - if the number goes beyond the available lines in the file, the cursor will be placed on the last line
- `gvim +/while script.py` opens `script.py` and the cursor is placed on the first line containing the given pattern
  - if the pattern is not found, the cursor will be placed on the last line
  - use `gvim +1 +/pattern` to force the search to start from the first line, otherwise cursor position stored in `viminfo` will be used (if applicable)

Execute command

- `gvim -c` allows you to execute the Command-line mode command passed as an argument
- `gvim -c '%s/search/replace/g' script.py` opens `script.py` and performs the given substitute operation
- `gvim -c 'normal =G' script.py` opens `script.py` and auto indents the entire file content

 As per :h -c, "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` option is similar to the `-c` option, but executes the command before loading any `vimrc` files.

Quickfix

- `gvim -q <(grep -Hn 'search' *.py)` interactively edit the matching lines from `grep` output
  - `-H` and `-n` options provide filename and line number prefix for the matching lines

    - use `:cn` and `:cp` to navigate to the next and previous occurrences respectively
    - Command-line area at the bottom will show the number of matches and filenames
    - you can also use `gvim -q file` if you had saved the `grep` output to that file
- `gvim -q error.log` edit source code based on compiler output containing filenames and line numbers for the error locations
    - here, the `error.log` is assumed to be the filename used to save the error messages

 See Vim and the quickfix list and stackoverflow: How do you use Vim's quickfix feature? to learn more about this feature.

 See :h quickfix for documentation.

Vimrc and Plugins

- `gvim -u file` uses the given file for initialization instead of `vimrc` files
    - useful to test plugins, apply a different `vimrc` based on which project you are working on, etc
- `gvim -u NONE` all initializations are skipped
- `gvim -u DEFAULTS` similar to `NONE`, but `defaults.vim` is loaded
- `gvim -u NORC` similar to `NONE`, but plugins are loaded
- `gvim --noplugin` only plugins are not loaded

Here's a neat table from :h --noplugin:

argument	vimrc	plugins	defaults.vim
(nothing)	yes	yes	yes
`-u NONE`	no	no	no
`-u DEFAULTS`	no	no	yes
`-u NORC`	no	yes	no
`--noplugin`	yes	no	yes

Session and Viminfo

- `gvim -S proj.vim` restore a session using the previously saved session file
    - see :h Session for more details
- `gvim -i proj.viminfo` restore Viminfo from the given file
    - this file will also be used instead of the default `viminfo` file to save information
    - see :h viminfo-read-write for more details