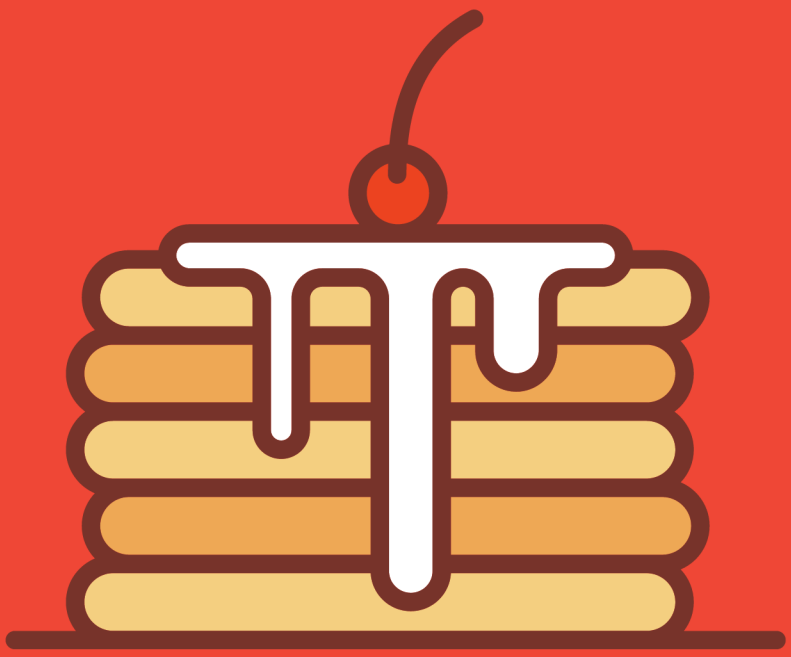


LEARN VIM

(The Smart Way)

Learning Vim and Vimscript isn't hard. This is the guide that you're looking for.

BY IGOR IRIANTO

Learn Vim

The Smart Way

Igor Irianto

This book is for sale at http://leanpub.com/learnvim

This version was published on 2021-02-18

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Igor Irianto by spreading the word about this book on Twitter!

The suggested hashtag for this book is #learnvim.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#learnvim

Contents

New To Vim? Read This First

Why This Guide Was Written

There are many places to learn Vim: the `vimtutor` is a great place to start and the `:help` manual has all the references you will ever need.

However, the average user needs something more than `vimtutor` and less than the `:help` manual. This guide attempts to bridge that gap by highlighting only the key features to learn the most useful parts of Vim in the least time possible.

Chances are you won't need all 100% of Vim features. You probably only need to know about 20% of them to become a powerful Vimmer. This guide will show you which Vim features you will find most useful.

This is an opinionated guide. It covers techniques that I often use when using Vim. The chapters are sequenced based on what I think would make the most logical sense for a beginner to learn Vim.

This guide is examples-heavy. When learning a new skill, examples are indispensable, having numerous examples will solidify these concepts more effectively.

Some of you may wonder why do you need to learn Vimscript? In my first year of using Vim, I was content with just knowing how to use Vim. Time passed and I started needing Vimscript more and more to write custom commands for my specific editing needs. As you are mastering Vim, you will sooner or later need to learn Vimscript. So why not sooner? Vimscript is a small language. You can learn its basics in just four chapters of this guide.

You can go far using Vim without knowing any Vimscript, but knowing it will help you excel even farther.

This guide is written for both beginner and advanced Vimmers. It starts out with broad and simple concepts and ends with specific and advanced concepts. If you're an advanced user already, I would encourage you to read this guide from start to finish anyway, because you will learn something new!

How To Transition To Vim From Using A Different Text Editor

Learning Vim is a satisfying experience, albeit hard. There are two main approaches to learn Vim:

1. Cold turkey
2. Gradual

Going cold turkey means to stop using whatever editor / IDE you were using and to use Vim exclusively starting now. The downside of this method is you will have a serious productivity loss during the first week or two. If you're a full-time programmer, this method may not be feasible. That's why for most people, I believe the best way to transition to Vim is to use it gradually.

To gradually use Vim, during the first two weeks, spend an hour a day using Vim as your editor while the rest of the time you can use other editors. Many modern editors come with Vim plugins. When I first started, I used VSCode's popular Vim plugin for an hour per day. I gradually increased the time with the Vim plugin until I finally used it all day. Keep in mind that these plugins can only emulate a fraction of Vim features. To experience the full power of Vim like Vimscript, Command-line (Ex) Commands, and external commands integration, you will need to use Vim itself.

There were two pivotal moments that made me start to use Vim 100%: when I grasped that Vim has a grammar-like structure (see chapter 4) and the fzf.vim[1] plugin (see chapter 3).

The first, when I realized Vim's grammar-like structure, was the defining moment that I finally understood what these Vim users were talking about. I didn't need to learn hundreds of unique commands. I only had to learn a small handful of commands and I could chain in a very intuitive way to do many things.

The second, the ability to quickly run a fuzzy file-search was the IDE feature that I used most. When I learned how to do that in Vim, I gained a major speed boost and never looked back ever since.

Everyone programs differently. Upon introspection, you will find that there are one or two features from your favorite editor / IDE that you use all the time. Maybe it was fuzzy-search, jump-to-definition, or quick compilation. Whatever they may be, identify them quickly and learn how to implement those in Vim (chances are Vim can probably do them too). Your editing speed will receive a huge boost.

Once you can edit at 50% of the original speed, it's time to go full-time Vim.

How To Read This Guide

This is a practical guide. To become good in Vim you need to develop your muscle memory, not head knowledge.

You don't learn how to ride a bike by reading a guide about how to ride a bike. You need to actually ride a bike.

You need to type along every commands referred in this guide. Not only that, but you need to repeat them several times and try different combinations. Look up what other features the command you just learned has. The `:help` command and search engines are your best friends. Your goal is not to know everything about a command, but to be able to execute that command naturally and instinctively.

[1] https://github.com/junegunn/fzf.vim

As much as I try to fashion this guide to be linear, some concepts in this guide have to be presented out-of-order. For example in chapter 1, I mention the substitute command (`:s`), even though it won't be covered until chapter 12. To remedy this, whenever a new concept that has not been covered yet is mentioned early, I will provide a quick how-to guide without a detailed explanation. So please bear with me :).

More Help

Here's one extra tip to use the help manual: suppose you want to learn more about what `Ctrl-P` does in insert mode. If you merely search for `:h CTRL-P`, you will be directed to normal mode's `Ctrl-P`. This is not the `Ctrl-P` help that you're looking for. In this case, search instead for `:h i_CTRL-P`. The appended `i_` represents the insert mode. Pay attention to which mode it belongs to.

Syntax

Most of the command or code-related phrases are in code-case (`like this`).

Strings are surrounded by a pair of double-quotes ("like this").

Vim commands can be abbreviated. For example, `:join` can be abbreviated as `:j`. Throughout the guide, I will be mixing the shorthand and the longhand descriptions. For commands that are not frequently used in this guide, I will use the longhand version. For commands that are frequently used, I will use the shorthand version. I apologize for the inconsistencies. In general, whenever you spot a new command, always check it on `:help` to see its abbreviations.

Vimrc

At various points in the guide, I will refer to vimrc options. If you're new to Vim, a vimrc is like a config file.

Vimrc won't be covered until chapter 21. For the sake of clarity, I will show briefly here how to set it up.

Suppose you need to set the number options (`set number`). If you don't have a vimrc already, create one. It is usually placed at the root directory named `.vimrc`. Depending on your OS, the location may differ. In macOS, I have it on `~/.vimrc`. To see where you should put yours, check out `:h vimrc`.

Inside it, add `set number`. Save it (`:w`), then source it (`:source %`). You should now see line numbers displayed on the left side.

Alternatively, if you don't want to a make permanent setting change, you can always run the `set` command inline, by running `:set number`. The downside of this approach is that this setting is temporary. When you close Vim, the option disappears.

Since we are learning about Vim and not Vi, a setting that you must have is the `nocompatible` option. Add `set nocompatible` in your vimrc. Many Vim-specific features are disabled when it is running on `compatible` option.

In general, whenever a passage mentions a vimrc option, just add that option into vimrc, save it, and source it.

Future, Errors, Questions

Expect more updates in the future. If you find any errors or have any questions, please feel free to reach out.

I also have planned a few more upcoming chapters, so stay tuned!

I Want More Vim Tricks

To learn more about Vim, please follow @learnvim[2].

Thank Yous

This guide wouldn't be possible without Bram Moleenar for creating Vim, my wife who had been very patient and supportive throughout the journey, all the contributors[3] of the learn-vim project, the Vim community, and many, many others that weren't mentioned.

Thank you. You all help make text editing fun :)

[2] https://twitter.com/learnvim
[3] https://github.com/iggredible/Learn-Vim/graphs/contributors

Ch01. Starting Vim

In this chapter, you will learn different ways to start Vim from the terminal. I was using Vim 8.2 when writing this guide. If you use Neovim or an older version of Vim, you should be mostly fine, but be aware that some commands might not be available.

Installing

I won't go through the detailed instruction how to install Vim in a specific machine. The good news is, most Unix-based computers should come with Vim installed. If not, most distros should have some instructions to install Vim.

For download informations, check out Vim's official download website or Vim's official github repository:

- Vim website[4]
- Vim github[5]

The Vim Command

Now that you have Vim installed, run this from the terminal:

```
vim
```

You should see an intro screen. This is the where you will be working on your file. Unlike most text editors and IDEs, Vim is a modal editor. If you want to type "hello", you need to switch to insert mode with `i`. Press `ihello<Esc>` to insert the text "hello".

Exiting Vim

There are several ways to exit Vim. The most common one is to type:

[4] https://www.vim.org/download.php
[5] https://github.com/vim/vim

```
:quit
```

You can type `:q` for short. That command is a command-line mode command (another one of Vim modes). If you type `:` in normal mode, the cursor will move to the bottom of the screen where you can type some commands. You will learn about the command-line mode later in chapter 15. If you are in insert mode, typing `:` will literally produce the character ":" on the screen. In this case, you need to switch back to normal mode. Type `<Esc>` to switch to normal mode. By the way, you can also return to normal mode from command-line mode by pressing `<Esc>`. You will notice that you can "escape" out of several Vim modes back to normal mode by pressing `<Esc>`.

Saving A File

To save your changes, type:

```
:write
```

You can also type `:w` for short. If this is a new file, you need to give it a name before you can save it. Let's name it `file.txt`. Run:

```
:w file.txt
```

To save and quit, you can combine the `:w` and `:q` commands:

```
:wq
```

To quit without saving any changes, add `!` after `:q` to force quit:

```
:q!
```

There are other ways to exit Vim, but these are the ones you will use daily.

Help

Throughout this guide, I will refer you to various Vim help pages. You can go to the help page by typing `:help {some-command}` (`:h` for short). You can pass to the `:h` command a topic or a command name as an argument. For example, to learn about different ways to quit Vim, type:

```
:h write-quit
```

How did I know to search for "write-quit"? I actually didn't. I just typed `:h`, then "quit", then `<Tab>`. Vim displayed relevant keywords to choose from. If you ever need to look up something ("I wish Vim can do this..."), just type `:h` and try some keywords, then `<Tab>`.

Opening a File

To open a file (`hello1.txt`) on Vim from the terminal, run:

```
vim hello1.txt
```

You can also open multiple files at once:

```
vim hello1.txt hello2.txt hello3.txt
```

Vim opens `hello1.txt`, `hello2.txt`, and `hello3.txt` in separate buffers. You will learn about buffers in the next chapter.

Arguments

You can pass the `vim` terminal command with different flags and options.

To check the current Vim version, run:

```
vim --version
```

This tells you the current Vim version and all available features marked with either `+` or `-` Some of these features in this guide require certain features to be available. For example, you will explore Vim's command-line history in a later chapter with the `:history` command. Your Vim needs to have `+cmdline_history` feature for the command to work. There is a good chance that the Vim you just installed have all the necessary features, especially if it is from a popular download source.

Many things you do from the terminal can also be done from inside Vim. To see the version from *inside* Vim, you can run this:

```
:version
```

If you want to open the file `hello.txt` and immediately execute a command, you can pass to the `vim` command the `+{cmd}` option.

In Vim, you can substitute texts with the `:s` command (short for `:substitute`). If you want to open `hello.txt` and substitute all "pancake" with "bagel", run:

```
vim +%s/pancake/bagel/g hello.txt
```

The command can be stacked:

```
vim +%s/pancake/bagel/g +%s/bagel/egg/g +%s/egg/donut/g hello.txt
```

Vim will replace all instances of “pancake” with “bagel”, then replace “bagel” with “egg”, then replace “egg” with “donut” (you willl learn substitution in a later chapter).

You can also pass the `c` flag followed by the command instead of the `+` syntax:

```
vim -c %s/pancake/bagel/g hello.txt
vim -c %s/pancake/bagel/g -c %s/bagel/egg/g -c %s/egg/donut/g hello.txt
```

Opening Multiple Windows

You can launch Vim on split horizontal and vertical windows with `o` and `O`, respectively.

To open Vim with two horizontal windows, run:

```
vim -o2
```

To open Vim with 5 horizontal windows, run:

```
vim -o5
```

To open Vim with 5 horizontal windows and fill up the first two with `hello1.txt` and `hello2.txt`, run:

```
vim -o5 hello1.txt hello2.txt
```

To open Vim with two vertical windows, 5 vertical windows, and 5 vertical windows with 2 files:

```
vim -O
vim -O5
vim -O5 hello1.txt hello2.txt
```

Suspending

If you need to suspend Vim while in the middle of editing, you can press `Ctrl-z`. You can also run either the `:stop` or `:suspend` command. To return to the suspended Vim, run `fg` from the terminal.

Starting Vim The Smart Way

You can pass the `vim` command with different options and flags, just like any terminal commands. One of the options is the command-line command (`+{cmd}` or `c cmd`). As you learn more commands throughout this guide, see if you can apply it on start. Also being a terminal command, you can combine `vim` with many other terminal commands. For example, you can redirect the output of the `ls` command to be edited in Vim with `ls -l | vim -`.

To learn more about Vim terminal command, check out `man vim`. To learn more about the Vim editor, continue reading this guide along with the `:help` command.

Ch02. Buffers, Windows, and Tabs

If you have used a modern text editor, you are probably familiar with windows and tabs. Vim uses three display abstractions instead of two: buffers, windows, and tabs. In this chapter, I will explain what buffers, windows, and tabs are and how they work in Vim.

Before you start, make sure you have the `set hidden` option in vimrc. Without it, whenever you switch buffers and your current buffer is not saved, Vim will prompt you to save the file (you don't want that if you want to move quickly). I haven't cover vimrc yet. If you don't have a vimrc, create one. It is usually placed at the root directory and named `.vimrc`. I have mine on `~/.vimrc`. To see where you should create your vimrc, check out `:h vimrc`. Inside it, add:

```
set hidden
```

Save it, then source it (run `:source %` from inside the vimrc).

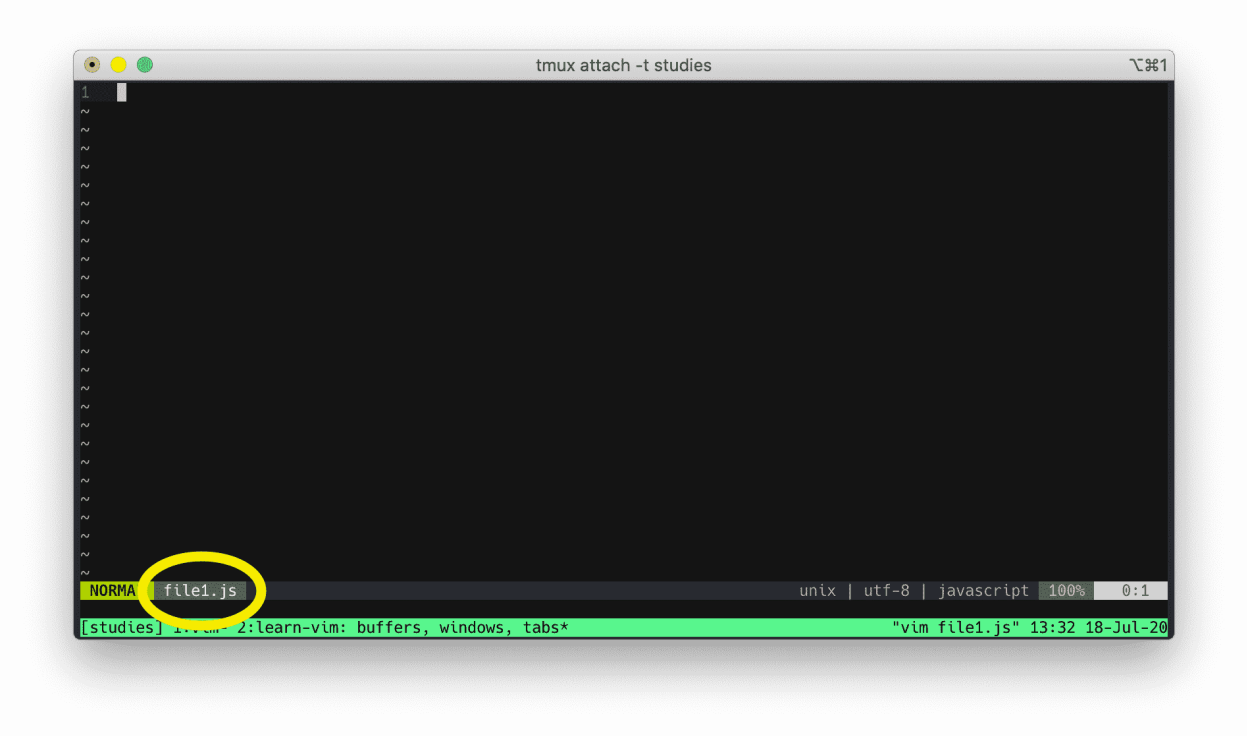
Buffers

What is a *buffer*?

A buffer is an in-memory space where you can write and edit some text. When you open a file in Vim, the data is bound to a buffer. When you open 3 files in Vim, you have 3 buffers.

Have two empty files, `file1.js` and `file2.j` (if possible, create them with Vim) available. Run this in the terminal:

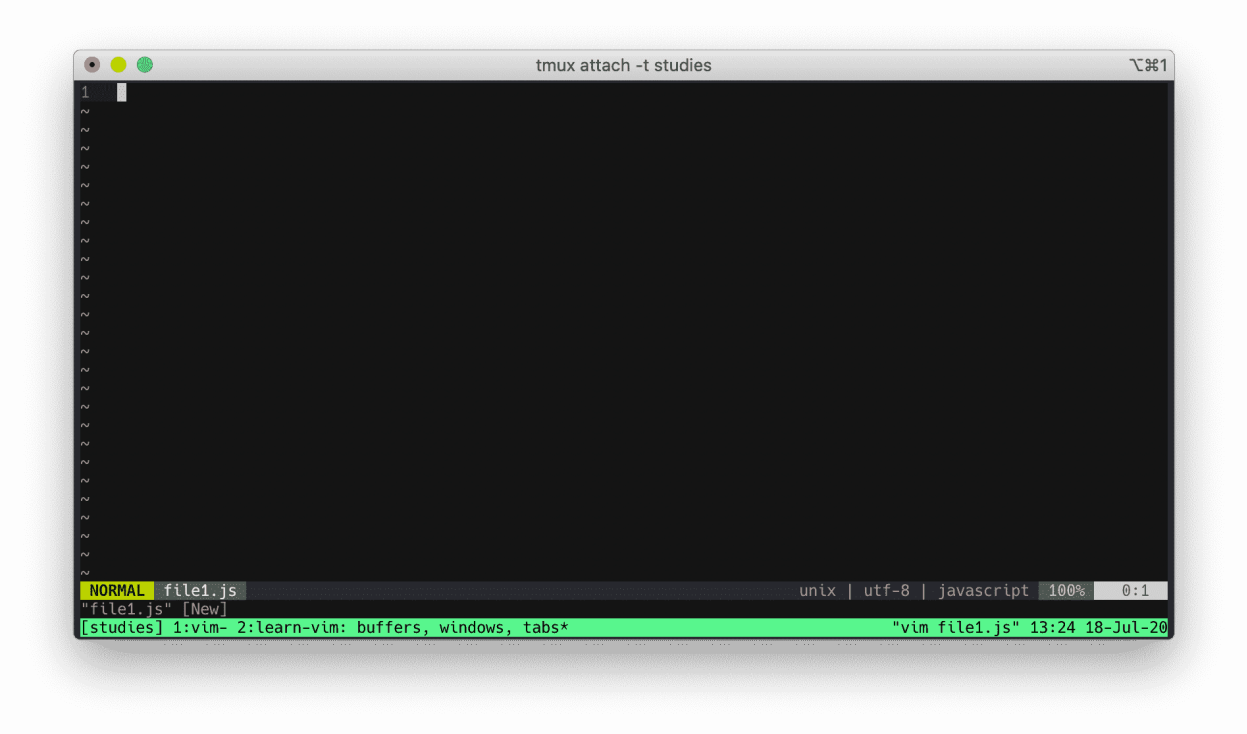
```
vim file1.js
```



one buffer displayed with highlight

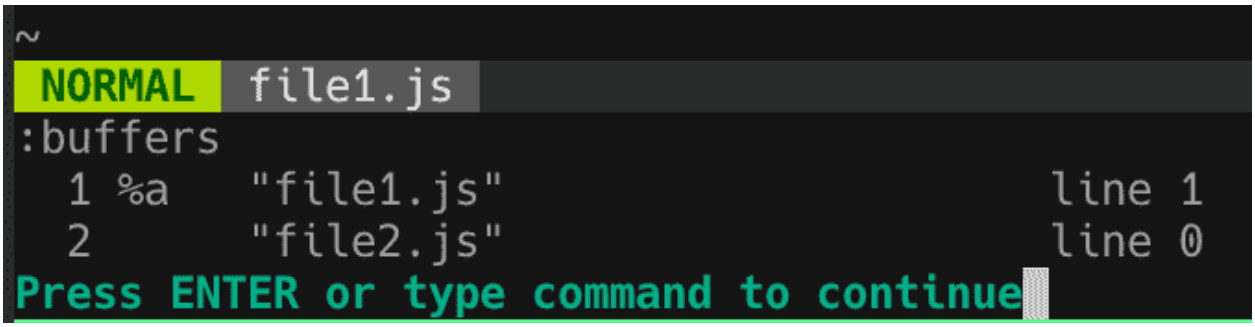
What you are seeing is `file1.js` *buffer*. Whenever you open a new file, Vim creates a new buffer. Exit Vim. This time, open two new files:

```
vim file1.js file2.js
```



one buffer displayed.png

Vim displays `file1.js` buffer, but it actually creates two buffers: `file1.js` buffer and `file2.js` buffer. Run `:buffers` to see all the buffers (alternatively, you can use `:ls` or `:files` too).



buffers command showing 2 buffers

There are several ways you can traverse buffers:

- `:bnext` to go to the next buffer (`:bprevious` to go to the previous buffer).
- `:buffer` + filename. Vim can autocomplete filename with `<Tab>`.
- `:buffer` + `n`, where `n` is the buffer number. For example, typing `:buffer 2` will take you to buffer #2.
- Jump to the older position in jump list with `Ctrl-O` and to the newer position with `Ctrl-I`. These are not buffer specific methods, but they can be used to jump between different buffers. I will talk more about jumps in Chapter 5.

- Go to the previously edited buffer with `Ctrl-^`.

Once Vim creates a buffer, it will remain in your buffers list. To remove it, you can type `:bdelete`. It accepts either a buffer number (`:bdelete 3` to delete buffer #3) or a filename (`:bdelete` then use `<Tab>` to autocomplete).

The hardest thing for me when learning about buffer was visualizing how buffers worked. Imagine a deck of playing cards. If I have 2 buffers, I have a stack of 2 cards. The card on top is the card I see. If I see `file1.js` buffer displayed then the `file1.js` card is on the top of the deck. I can't see the other card, `file2.js`. If I switch buffers to `file2.js`, that `file2.js` card is now on the top of the deck and `file1.js` card is at the bottom.

If you haven't used Vim before, this is a new concept. Take your time to understand it.

Exiting Vim

By the way, if you have multiple buffers opened, you can close all of them with quit-all:

```
:qall
```

If you want to close without saving your changes, just add `!` at the end:

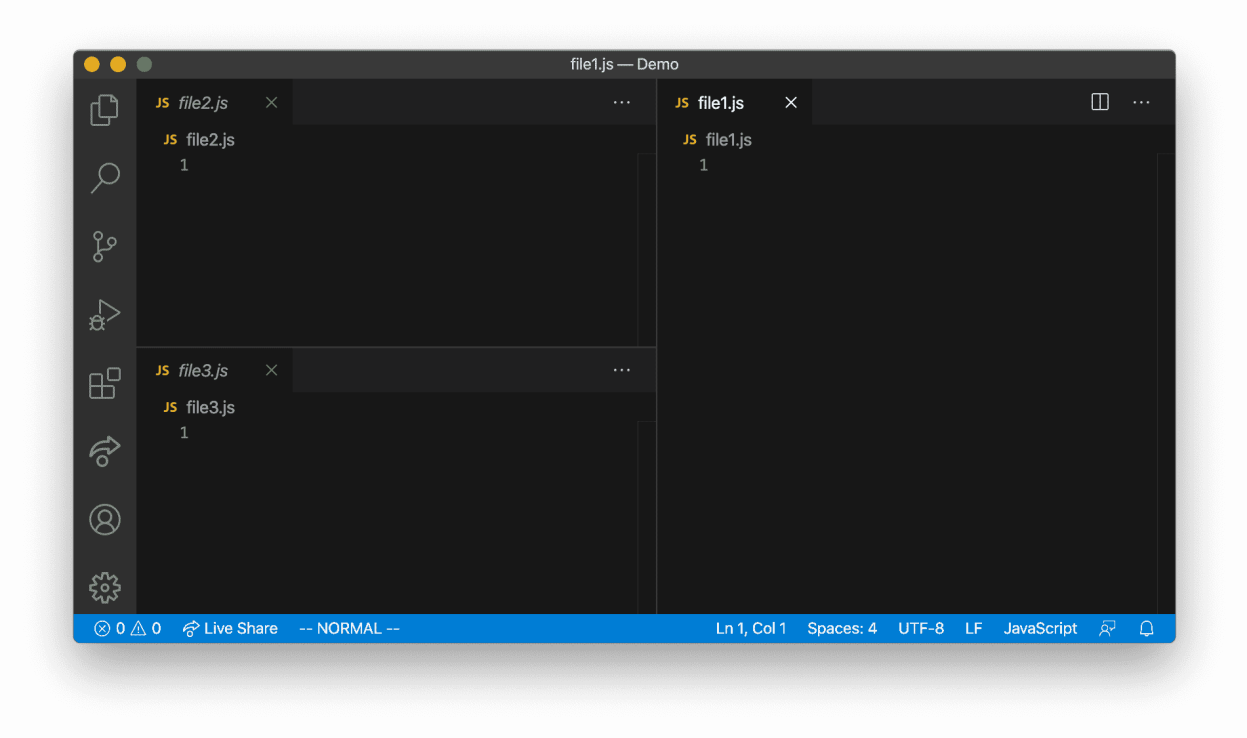
```
:qall!
```

To save and quit all, run:

```
:wqall
```

Windows

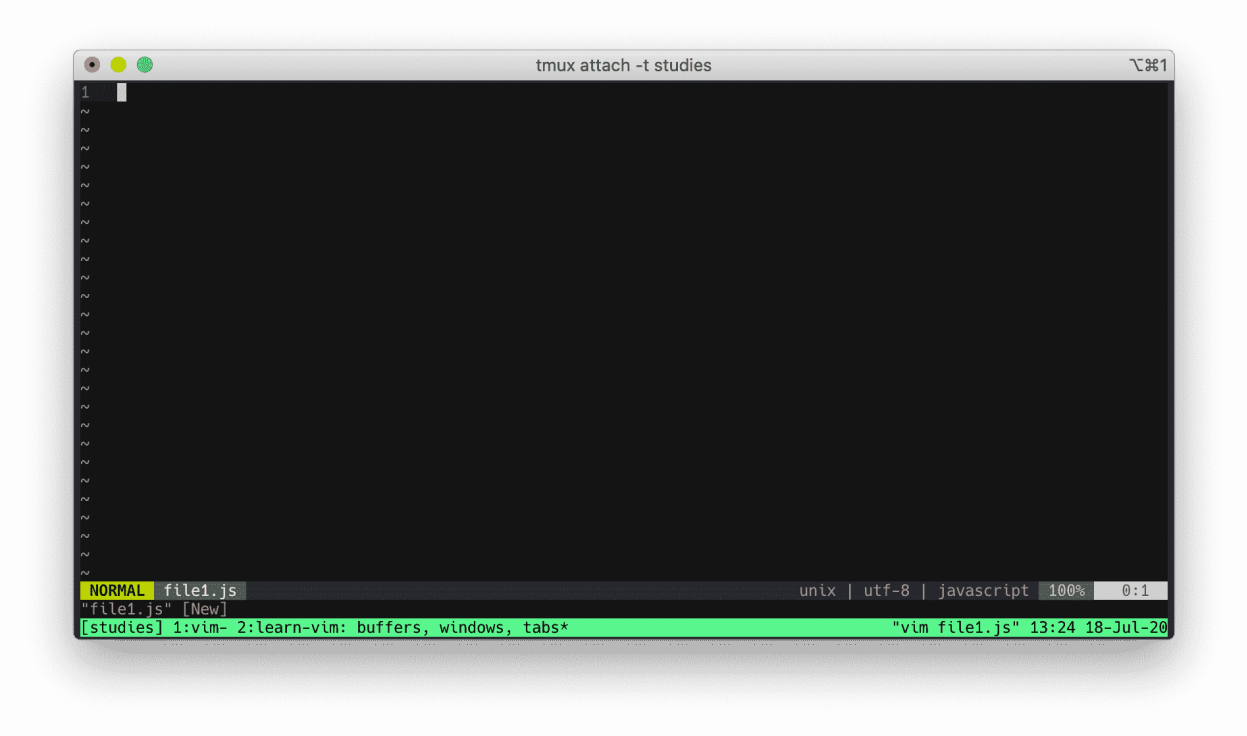
A window is a viewport on a buffer. You can have multiple windows. Most text editors have the ability to display multiple windows. Below you see a VSCode with 3 windows:



VSCode showing 3 windows

Let's open `file1.js` from the terminal again:

```
vim file1.js
```

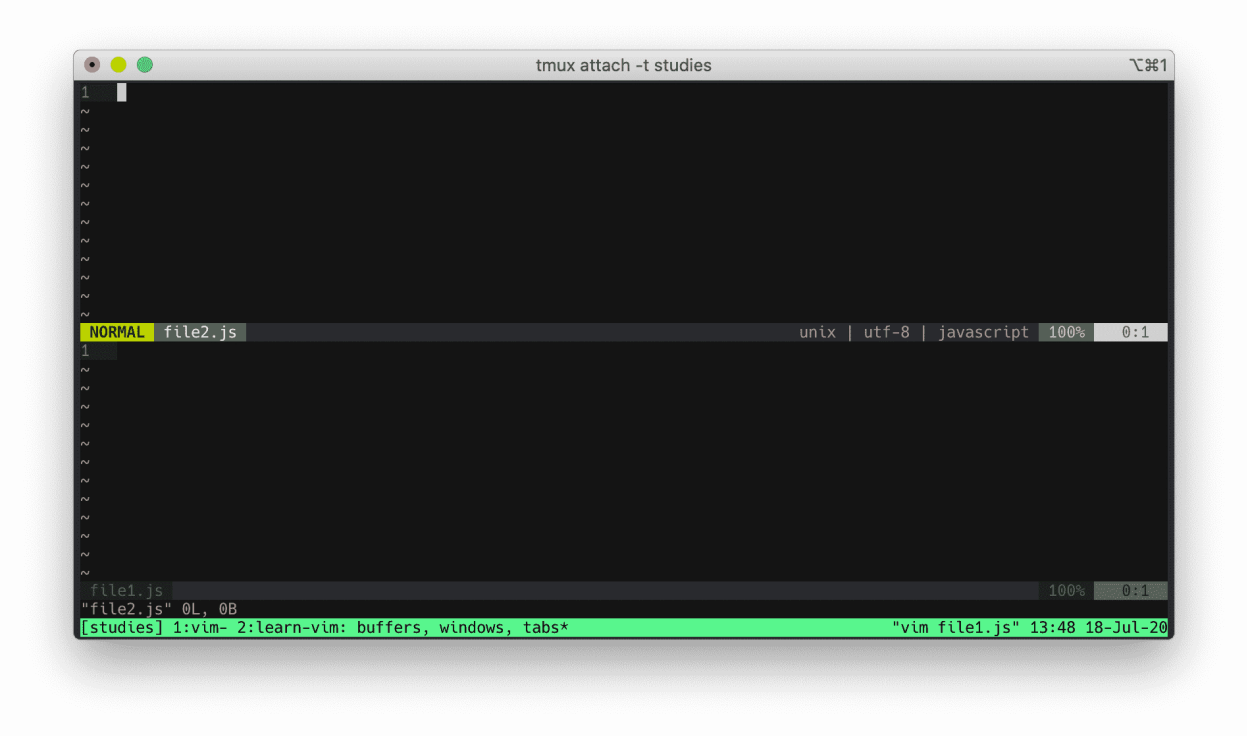


one buffer displayed.png

Earlier I said that you're looking at `file1.js` buffer. While that was correct, it was incomplete. You are looking at `file1.js` buffer displayed through **a window**. A window is what you are seeing a buffer through.

Don't quit Vim yet. Run:

```
:split file2.js
```



split window horizontally

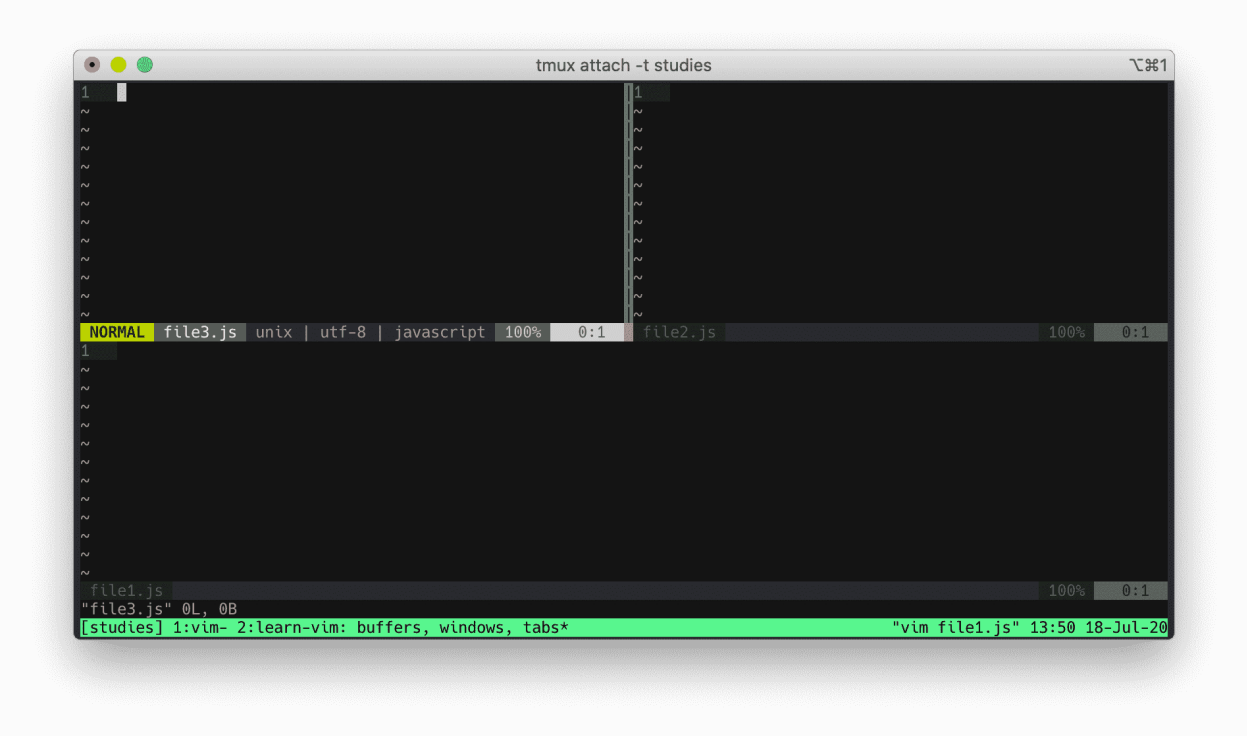
Now you are looking at two buffers through **two windows**. The top window displays `file2.js` buffer. The bottom window displays `file1.js` buffer.

If you want to navigate between windows, use these shortcuts:

```
Ctrl-W H    Moves the cursor to the left window
Ctrl-W J    Moves the cursor to the window below
Ctrl-W K    Moves the cursor to the window upper
Ctrl-W L    Moves the cursor to the right window
```

Now run:

```
:vsplit file3.js
```

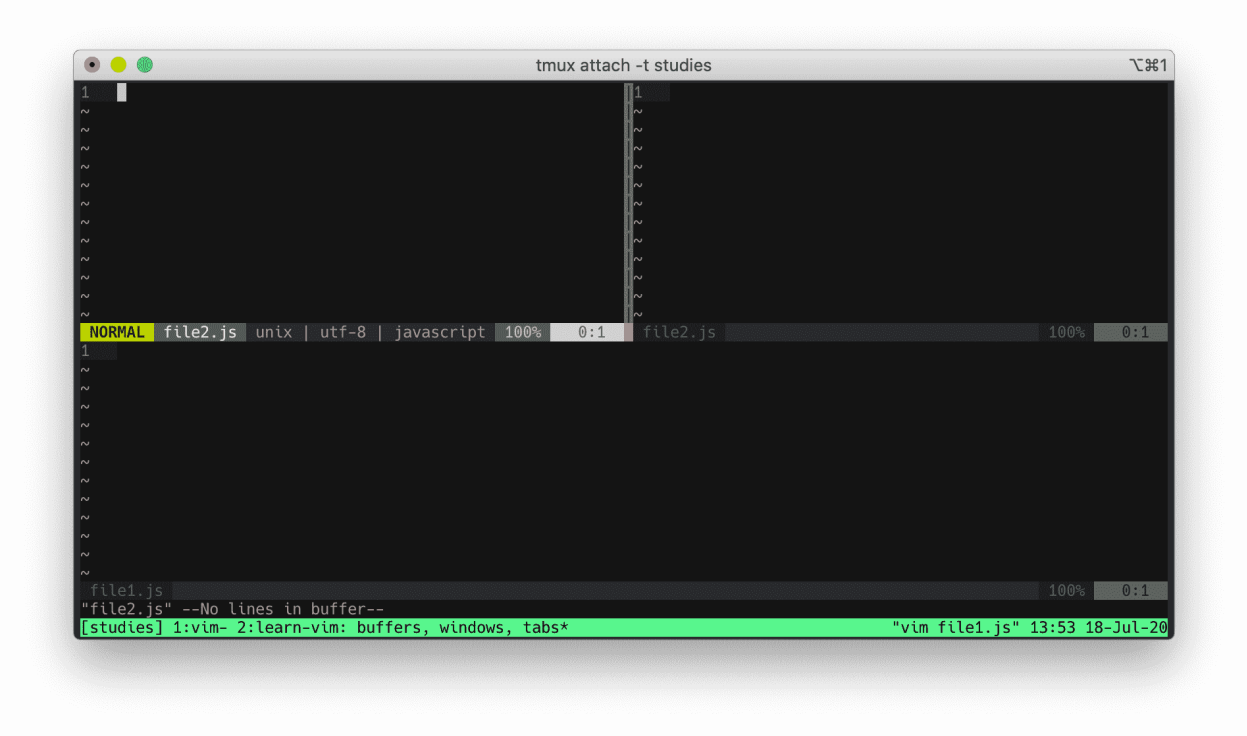


split window vertically and horizontally

You are now seeing three windows displaying three buffers. The top left window displays `file3.js` buffer, the top right window displays `file2.js` buffer, and the bottom window displays `file1.js` buffer.

You can have multiple windows displaying the same buffer. While you're on the top left window, type:

```
:buffer file2.js
```



split window vertically and horizontally with two file2.js

Now both top left and top right windows are displaying `file2.js` buffer. If you start typing on the top left, you can see that the content on both top left and top right window are being updated in real-time.

To close the current window, you can run `Ctrl-W C` or type `:quit`. When you close a window, the buffer will still be there (run `:buffers` to confirm this).

Here are some useful normal-mode window commands:

```
Ctrl-W V    Opens a new vertical split
Ctrl-W S    Opens a new horizontal split
Ctrl-W C    Closes a window
Ctrl-W O    Makes the current window the only one on screen and closes other windows
```

And here is a list of useful window command-line commands:

```
:vsplit filename    Split window vertically
:split filename     Split window horizontally
:new filename       Create new window
```

Take your time to understand them. For more, check out `:h window`.

Tabs

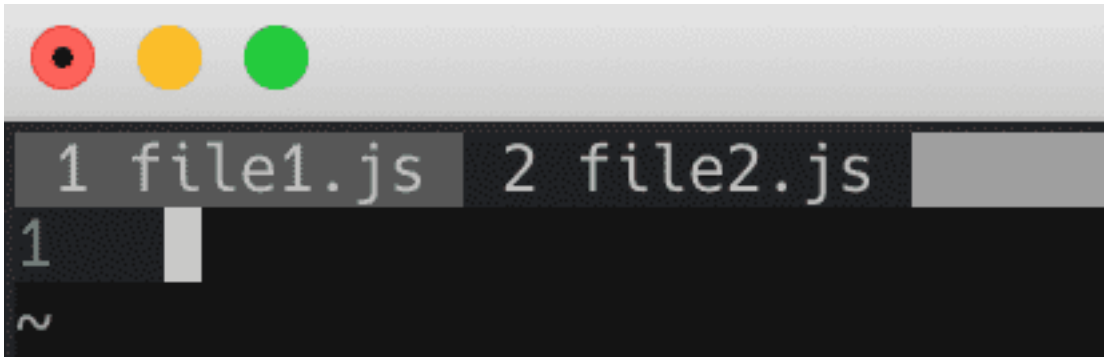
A tab is a collection of windows. Think of it like a layout for windows. In most modern text editors (and modern internet browsers), a tab means an open file / page and when you close it, that file / page goes away. In Vim, a tab does not represent an open file. When you close a tab in Vim, you are not closing a file. You are only closing the layout. The data for those files are stored in-memory in buffers. The buffers are still there.

Let's see Vim tabs in action. Open `file1.js`:

```
vim file1.js
```

To open `file2.js` in a new tab:

```
:tabnew file2.js
```



screen displays tab 2

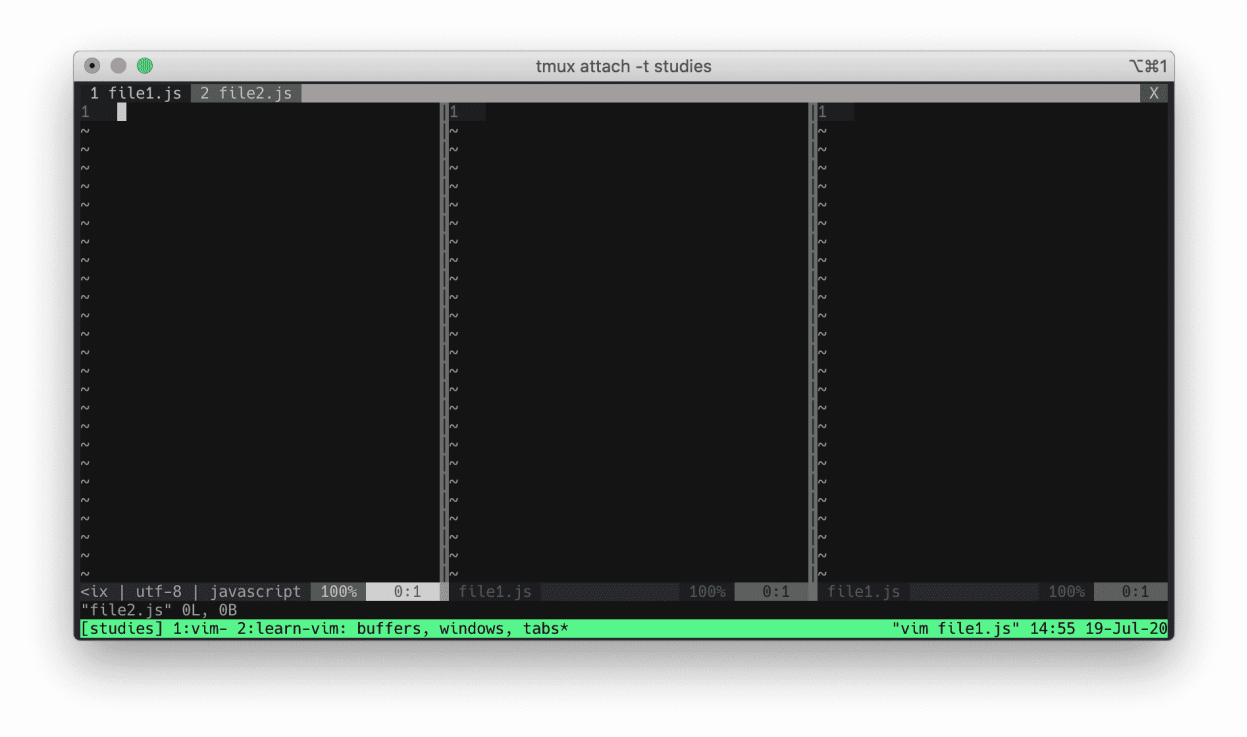
You can also let Vim autocomplete the file you want to open in a *new tab* by pressing `<Tab>` (no pun intended).

Below is a list of useful tab navigations:

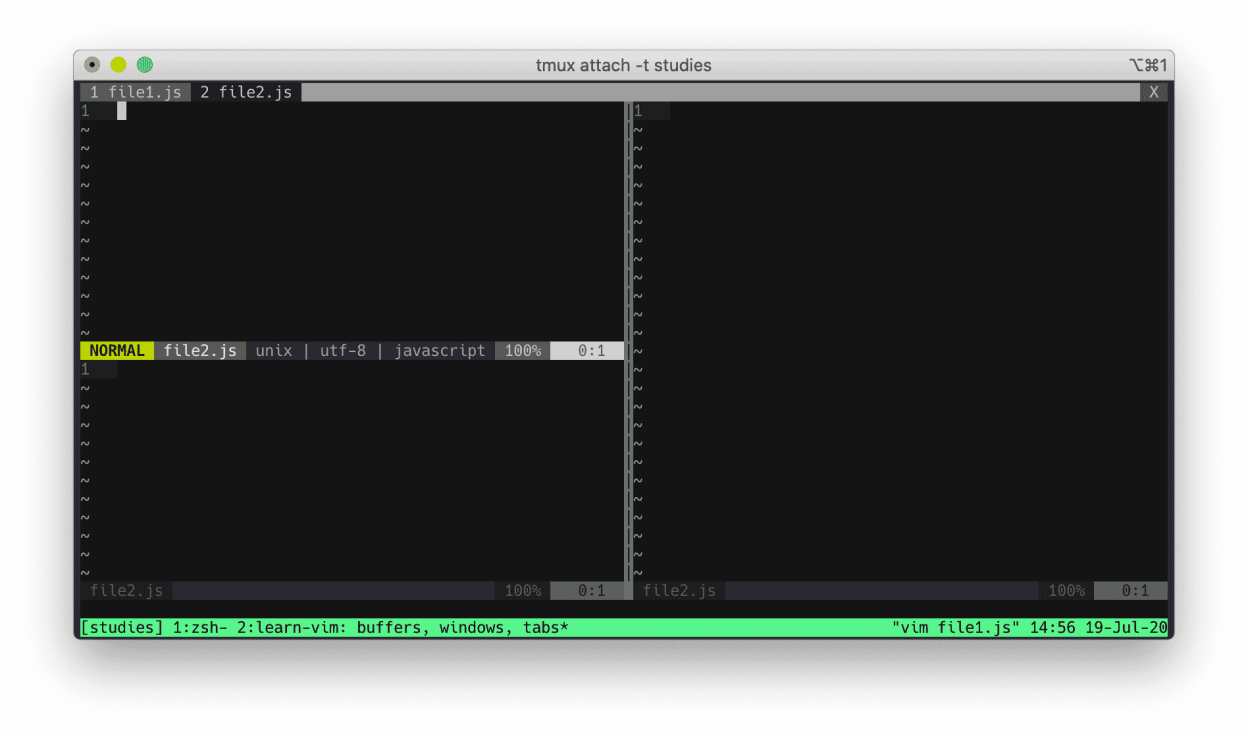
```
:tabnew file.txt    Open file.txt in a new tab
:tabclose           Close the current tab
:tabnext            Go to next tab
:tabprevious        Go to previous tab
:tablast            Go to last tab
:tabfirst           Go to first tab
```

You can also run `gt` to go to next tab page (you can go to previous tab with `gT`). You can pass count as argument to `gt`, where count is tab number. To go to the third tab, do `3gt`.

One advantage of having multiple tabs is you can have different window arrangements in different tabs. Maybe you want your first tab to have 3 vertical windows and second tab to have a mixed horizontal and vertical windows layout. Tab is the perfect tool for the job!



first tab with multiple windows



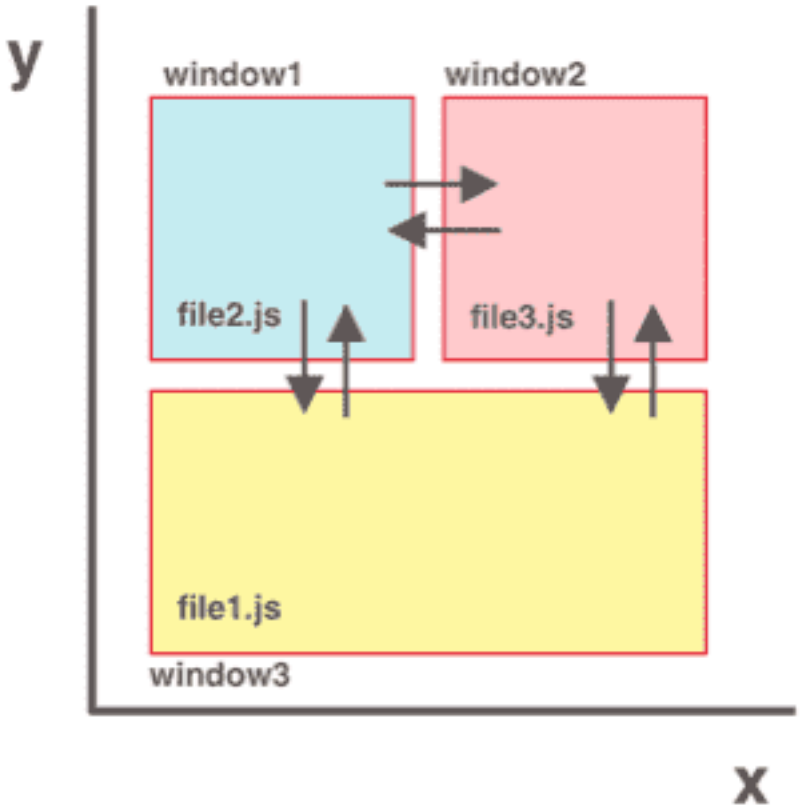
second tab with multiple windows

To start Vim with multiple tabs, you can do this from the terminal:

```
vim -p file1.js file2.js file3.js
```

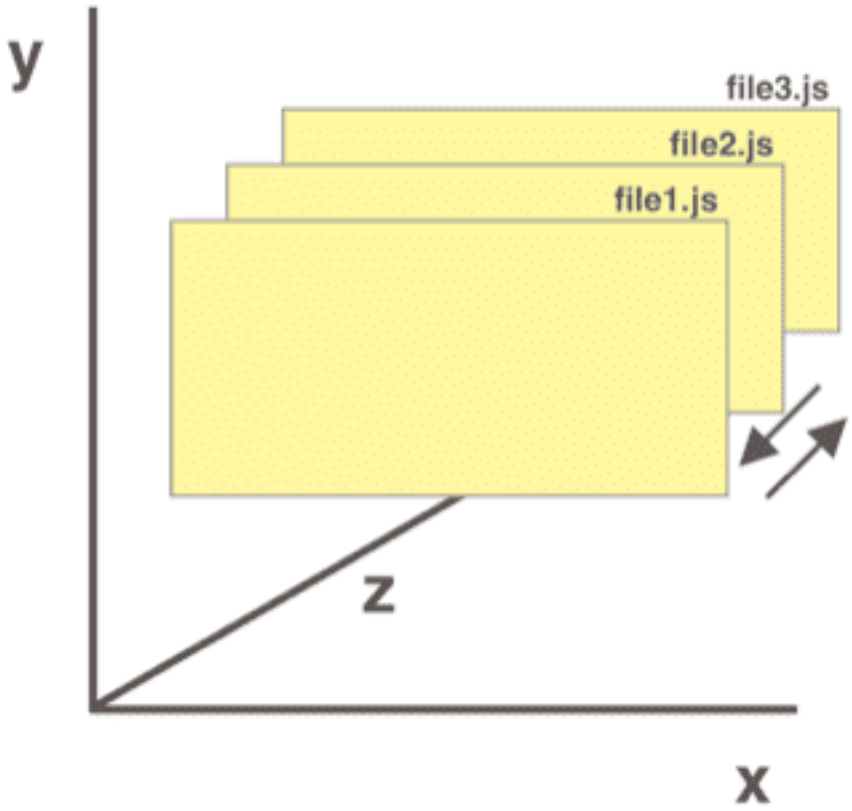
Moving In 3D

Moving between windows is like traveling two-dimensionally along X-Y axis in a Cartesian coordinate. You can move to the top, right, bottom, and left window with `Ctrl-W H/J/K/L`.



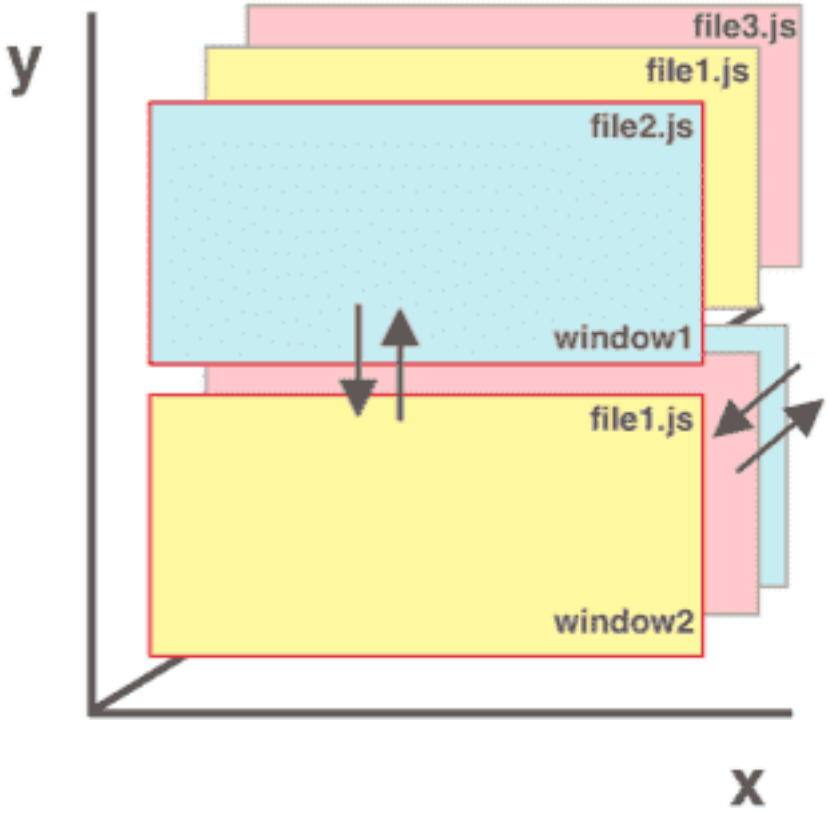
cartesian movement in x and y axis

Moving between buffers is like traveling across the Z axis in a Cartesian coordinate. Imagine your buffer files lining up across the Z axis. You can traverse the Z axis one buffer at a time with `:bnext` and `:bprevious`. You can jump to any coordinate in Z axis with `:buffer filename/buffernumber`.



cartesian movement in z axis

You can move in *three-dimensional space* by combining window and buffer movements. You can move to the top, right, bottom, or left window (X-Y navigations) with window navigations. Since each window contains buffers, you can move forward and backward (Z navigations) with buffer movements.



cartesian movement in x, y, and z axis

Using Buffers, Windows, and Tabs The Smart Way

You have learned what buffers, windows, and tabs are and how they work in Vim. Now that you understand them better, you can use them in your own workflow.

Everyone has a different workflow, here is mine for example:

- First I use buffers to store all the required files for the current task. Vim can handle many open buffers before it starts slowing down. Plus having many buffers opened won't crowd my screen. I am only seeing one buffer (assuming I have only one window) at any time, allowing me to focus on one screen. When I need to go somewhere, I can quickly fly to any open buffer anytime.
- I use multiple windows to view multiple buffers at once, usually when diffing files, reading docs, or following a code flow. I try to keep the number of windows opened to no more than

three because my screen will get crowded (I use a small laptop). When I am done, I close any extra windows. Fewer windows means less distractions.
- Instead of tabs, I use tmux[6] windows. I usually use multiple tmux windows at once. For example, one tmux window for client-side codes and another for backend codes.

My workflow may look different than yours based on your editing style and that's fine. Experiment around to discover your own flow suited for your coding style.

[6] https://github.com/tmux/tmux/wiki

Ch03. Searching Files

The goal of this chapter is to introduce you to how to search quickly in Vim. Being able to search quickly is a great way to jump-start your Vim productivity. When I figured out how to search files quickly, I made the switch to use Vim full-time.

This chapter is divided into two parts: how to search without plugins and how to search with fzf.vim[7] plugin. Let's get started!

Opening And Editing Files

To open a file in Vim, you can use `:edit`.

```
:edit file.txt
```

If `file.txt` exists, it opens the `file.txt` buffer. If `file.txt` doesn't exist, it creates a new buffer for `file.txt`.

Autocomplete with `<Tab>` works with `:edit`. For example, if your file is inside a Rails[8] *a*pp *c*ontroller *u*sers controller directory `./app/controllers/users_controllers.rb`, you can use `<Tab>` to expand the terms quickly:

```
:edit a<Tab>c<Tab>u<Tab>
```

`:edit` accepts wildcards arguments. `*` matches any file in the current directory. If you are only looking for files with `.yml` extension in the current directory:

```
:edit *.yml<Tab>
```

Vim will give you a list of all `.yml` files in the current directory to choose from.

You can use `**` to search recursively. If you want to look for all `*.md` files in your project, but you are not sure in which directories, you can do this:

```
:edit **/*.md<Tab>
```

`:edit` can be used to run `netrw`, Vim's built-in file explorer. To do that, give `:edit` a directory argument instead of file:

[7] https://github.com/junegunn/fzf.vim
[8] https://rubyonrails.org/

```
:edit .
:edit test/unit/
```

Searching Files With Find

You can find files with `:find`. For example:

```
:find package.json
:find app/controllers/users_controller.rb
```

Autocomplete also works with `:find`:

```
:find p<Tab>                " to find package.json
:find a<Tab>c<Tab>u<Tab>    " to find app/controllers/users_controller.rb
```

You may notice that `:find` looks like `:edit`. What's the difference?

Find And Path

The difference is that `:find` finds file in `path`, `:edit` doesn't. Let's learn a little bit about this `path`. Once you learn how to modify your paths, `:find` can become a powerful searching tool. To check what your paths are, do:

```
:set path?
```

By default, yours probably look like this:

```
path=.,/usr/include,,
```

- `.` means to search relative to the directory of the current file.
- `,` means to search in the current directory.
- `/usr/include` is the directory for C compilers header files.

The first two are important and the third one can be ignored for now. The take-home here is that you can modify your own paths. Let's assume this is your project structure:

```
app/
  assets/
  controllers/
    application_controller.rb
    comments_controller.rb
    users_controller.rb
    ...
```

If you want to go to `users_controller.rb` from the root directory, you have to go through several directories (and pressing a considerable amount of tabs). Often when working with a framework, you spend 90% of your time in a particular directory. In this situation, you only care about going to the `controllers/` directory with the least amount of keypress. The `path` setting can shorten that journey.

You need to add the `app/controllers/` to the current `path`. Here is how you can do it:

```
:set path+=app/controllers/
```

Now that your path is updated, when you type `:find u<Tab>`, Vim will now search inside `app/controllers/` directory for files starting with "u".

If you have a nested `controllers/` directory, like `app/controllers/account/users_controller.rb`, Vim won't find `users_controllers`. You need to instead add `:set path+=app/controllers/**` so autocomplete will find `users_controller.rb`. This is great! Now you can find the users controller with 1 press of tab instead of 3.

You might be thinking to add the entire project directories so when you press `tab`, Vim will search everywhere for that file, like this:

```
:set path+=$PWD/**
```

`$PWD` is the current working directory. If you try to add your entire project to `path` hoping to make all files to be reachable upon a `tab` press, although this may work for a small project, doing this will slow down your search significantly if you have a large number of files in your project. I recommend adding only the `path` of your most visited files / directories.

You can add the `set path+={your-path-here}` in your vimrc. Updating `path` takes only a few seconds and doing this will save you a lot of time.

Searching In Files With Grep

If you need to find in files (find phrases in files), you can use grep. Vim has two ways of doing that:

- Internal grep (`:vim`. Yes, it is spelled `:vim`. It is short for `:vimgrep`).
- External grep (`:grep`).

Let's go through internal grep first. `:vim` has the following syntax:

```
:vim /pattern/ file
```

- `/pattern/` is a regex pattern of your search term.
- `file` is the file arguments. You can pass multiple arguments. Vim will search for the pattern inside the file arguments. Similar to `:find`, you can pass it `*` and `**` wildcards.

For example, to look for all occurrences of "breakfast" string inside all ruby files (`.rb`) inside `app/controllers/` directory:

```
:vim /breakfast/ app/controllers/**/*.rb
```

After running that, you will be redirected to the first result. Vim's `vim` search command uses `quickfix` operation. To see all search results, run `:copen`. This opens a `quickfix` window. Here are some useful quickfix commands to get you productive immediately:

```
:copen        Open the quickfix window
:cclose       Close the quickfix window
:cnext        Go to the next error
:cprevious    Go to the previous error
:colder       Go to the older error list
:cnewer       Go to the newer error list
```

To learn more about quickfix, check out `:h quickfix`.

You may notice that running internal grep (`:vim`) can get slow if you have a large number of matches. This is because Vim reads the searches into memory. Vim loads each matching files as if they are being edited. If Vim checks a large number of files, it will consume a large amount of memory.

Let's talk about external grep. By default, it uses `grep` terminal command. To search for "lunch" inside a ruby file inside `app/controllers/` directory, you can do this:

```
:grep -R "lunch" app/controllers/
```

Note that instead of using `/pattern/`, it follows the terminal grep syntax `"pattern"`. It also displays all matches using `quickfix`.

Vim uses `grepprg` variable to determine which external program to run when running `:grep` so you don't have to always use the terminal `grep` command. Later I will show you how to change default the grep external program.

Browsing Files With Netrw

`netrw` is Vim's built-in file explorer. It is useful to see a project's structural hierarchy. To run `netrw`, you need these two settings in your `.vimrc`:

```
set nocp
filetype plugin on
```

Since `netrw` is a vast topic, I will only cover the basic usage, but it should be enough to get you started. You can start `netrw` when you launch Vim and passing it a directory instead of a file. For example:

```
vim .
vim src/client/
vim app/controllers/
```

To launch `netrw` from inside Vim, you can use the `:edit` command and pass it a directory instead of a filename:

```
:edit .
:edit src/client/
:edit app/controllers/
```

There are other ways to launch `netrw` window without passing a directory:

```
:Explore     Starts netrw on current file
:Sexplore    No kidding. Starts netrw on split top half of the screen
:Vexplore    Starts netrw on split left half of the screen
```

You can navigate `netrw` with Vim motions (motions will be covered in depth in a later chapter). If you need to create, delete, and rename a directory, here is a list of useful `netrw` commands:

```
d    Create a new directory
R    Rename a file or directory
D    Delete a file or directory
```

`:h netrw` is very comprehensive. Check it out if you have time.

If you find `netrw` too bland and need more flavor, vim-vinegar[9] is a good plugin to improve `netrw`. If you're looking for a different file explorer, NERDTree[10] is a good alternative. Check them out!

Fzf

Now that you've learned how to search files in Vim with built-in tools, let's learn how to do it with plugins.

One thing that modern text editors get right that Vim didn't is how easy it is to find files and to find in files using fuzzy search. In this second half of the chapter, I will show you how to use fzf.vim[11] to make searching in Vim easy and powerful.

[9] https://github.com/tpope/vim-vinegar
[10] https://github.com/preservim/nerdtree
[11] https://github.com/junegunn/fzf.vim

Setup

First, make sure you have fzf[12] and ripgrep[13] downloaded. Follow the instruction on their github repo. The commands `fzf` and `rg` should now be available after successful installs.

Ripgrep is a search tool much like grep (hence the name). It is generally faster than grep and has many useful features. Fzf is a general-purpose command-line fuzzy finder. You can use it with any commands, including ripgrep. Together, they make a powerful search tool combination.

Fzf does not use ripgrep by default, so we need to tell fzf to use ripgrep with `FZF_DEFAULT_COMMAND` variable. In my `.zshrc` (`.bashrc` if you use bash), I have these:

```
if type rg &> /dev/null; then
  export FZF_DEFAULT_COMMAND='rg --files'
  export FZF_DEFAULT_OPTS='-m'
fi
```

Pay attention to `-m` in `FZF_DEFAULT_OPTS`. This option allows us to make multiple selections with `<Tab>` or `<Shift-Tab>`. You don't have to have this line to make fzf to work with Vim, but I think it is a useful option to have. It will come in handy when you want to perform search and replace in multiple files which I'll cover in just a little bit. The fzf command accepts many more flags, but I won't cover them here. To learn more, check out fzf's repo[14] or `man fzf`. At minimum you should have `export FZF_DEFAULT_COMMAND='rg'`.

After installing fzf and ripgrep, let's set up the fzf plugin. I am using vim-plug[15] plugin manager in this example, but you can use any plugin managers.

Add these inside your `.vimrc` plugins. You need to use fzf.vim[16] plugin (created by the same fzf author).

```
Plug 'junegunn/fzf.vim'
Plug 'junegunn/fzf', { 'do': { -> fzf#install() } }
```

For more info about this plugin, you can check out fzf.vim repo[17].

Fzf Syntax

To use fzf efficiently, you should learn some basic fzf syntax. Fortunately, the list is short:

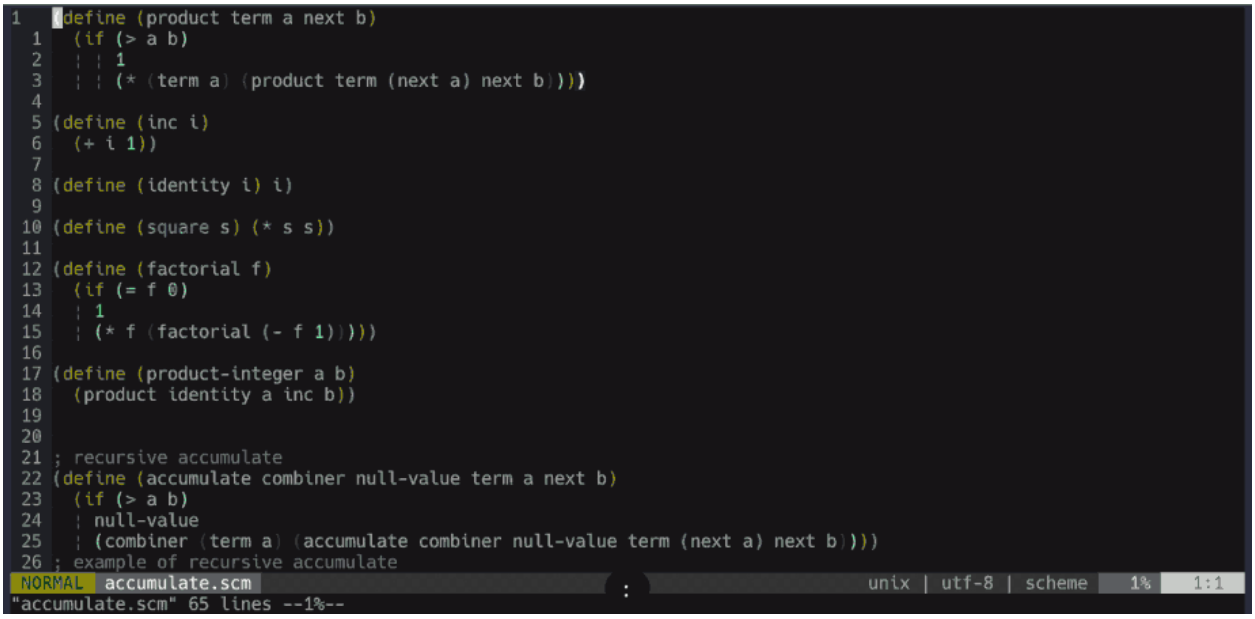
[12] https://github.com/junegunn/fzf
[13] https://github.com/BurntSushi/ripgrep
[14] https://github.com/junegunn/fzf#usage
[15] https://github.com/junegunn/vim-plug
[16] https://github.com/junegunn/fzf.vim
[17] https://github.com/junegunn/fzf/blob/master/README-VIM.md

- `^` is a prefix exact match. To search for a phrase starting with "welcome": `^welcome`.
- `$` is a suffix exact match. To search for a phrase ending with "my friends": `friends$`.
- `'` is an exact match. To search for the phrase "welcome my friends": `'welcome my friends`.
- `|` is an "or" match. To search for either "friends" or "foes": `friends | foes`.
- `!` is an inverse match. To search for phrase containing "welcome" and not "friends": `welcome !friends`

You can mix and match these options. For example, `^hello | ^welcome friends$` will search for the phrase starting with either "welcome" or "hello" and ending with "friends".

Finding Files

To search for files inside Vim using fzf.vim plugin, you can use the `:Files` method. Run `:Files` from Vim and you will be prompted with fzf search prompt.



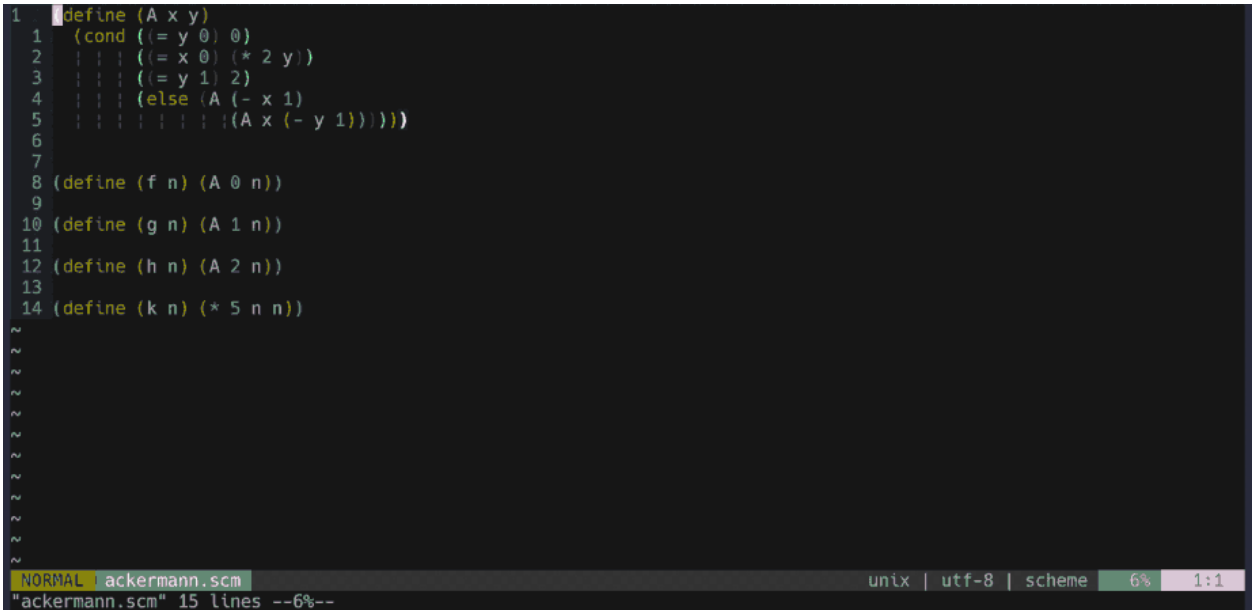
Finding files in fzf

Since you will be using this command frequently, it is good to have this mapped. I map mine to `Ctrl-f`. In my vimrc, I have this:

```
nnoremap <silent> <C-f> :Files<CR>
```

Finding In Files

To search inside files, you can use the `:Rg` command.



Finding in files in fzf

Again, since you will probably use this frequently, let's map it. I map mine to `<Leader>f`.

```
nnoremap <silent> <Leader>f :Rg<CR>
```

Other Searches

Fzf.vim provides many other search commands. I won't go through each one of them here, but you can check them out here[18].

Here's what my fzf maps look like:

```
nnoremap <silent> <Leader>b :Buffers<CR>
nnoremap <silent> <C-f> :Files<CR>
nnoremap <silent> <Leader>f :Rg<CR>
nnoremap <silent> <Leader>/ :BLines<CR>
nnoremap <silent> <Leader>' :Marks<CR>
nnoremap <silent> <Leader>g :Commits<CR>
nnoremap <silent> <Leader>H :Helptags<CR>
nnoremap <silent> <Leader>hh :History<CR>
nnoremap <silent> <Leader>h: :History:<CR>
nnoremap <silent> <Leader>h/ :History/<CR>
```

[18] https://github.com/junegunn/fzf.vim#commands

Replacing Grep With Rg

As mentioned earlier, Vim has two ways to search in files: `:vim` and `:grep`. `:grep` uses external search tool that you can reassign using the `grepprg` keyword. I will show you how to configure Vim to use ripgrep instead of terminal grep when running the `:grep` command.

Now let's setup `grepprg` so `:grep` uses ripgrep. Add this in your vimrc:

```
set grepprg=rg\ --vimgrep\ --smart-case\ --follow
```

Feel free to modify some of the options above! For more information what the options above mean, check out `man rg`.

After you updated `grepprg`, now when you run `:grep`, it runs `rg --vimgrep --smart-case --follow` instead of `grep`. If you want to search for "donut" using ripgrep, you can now run a more succinct command `:grep "donut"` instead of `:grep "donut" . -R`

Just like the old `:grep`, this new `:grep` also uses quickfix to display results.

You might wonder, "Well, this is nice but I never used `:grep` in Vim, plus can't I just use `:Rg` to find phrases in files? When will I ever need to use `:grep`?

That is a very good question. You may need to use `:grep` in Vim to do search and replace in multiple files, which I will cover next.

Search And Replace In Multiple Files

Modern text editors like VSCode makes it very easy to search and replace a string across multiple files. In this section, I will show you two different methods to easily do that in Vim.

The first method is to replace *all* matching phrases in your project. You will need to use `:grep`. If you want to replace all instances of "pizza" with "donut", here's what you do:

```
:grep "pizza"
:cfdo %s/pizza/donut/g | update
```

Let's break down the commands:

1. `:grep pizza` uses ripgrep to search for all instances of "pizza" (by the way, this would still work even if you didn't reassign `grepprg` to use ripgrep. You would have to do `:grep "pizza" . -R` instead of `:grep "pizza"`).
2. `:cfdo` executes any command you pass to all files in your quickfix list. In this case, your command is the substitution command `%s/pizza/donut/g`. The pipe (`|`) is a chain operator. The `updat` command saves each file after substitution. I will cover substitute command in more depth in a later chapter.

The second method is to search and replace in select files. With this method, you can manually choose which files you want to perform select and replace on. Here is what you do:

1. Clear your buffers first. It is imperative that your buffer list contains only the files you need. You can either restart Vim or run `:%bd | e#` command (`%bd` deletes all the buffers and `e#` opens the file you were just on).
2. Run `:Files`.
3. Select all files you want to perform search-and-replace on. To select multiple files, use `<Tab>` / `<Shift-Tab>`. This is only possible if you have the multiple flag (`-m`) in `FZF_DEFAULT_OPTS`.
4. Run `:bufdo %s/pizza/donut/g | update`. The command `:bufdo %s/pizza/donut/g | update` looks similar to the earlier `:cfdo %s/pizza/donut/g | update` command. The difference is instead of substituting all quickfix entries (`:cfdo`), you are substituting all buffer entries (`:bufdo`).

Learn Search The Smart Way

Searching is the bread-and-butter of text editing. Learning how to search well in Vim will improve your text editing workflow significantly.

Fzf.vim is a game-changer. I can't imagine using Vim without it. I think it is very important to have a good search tool when starting Vim. I've seen people struggling to transition to Vim because it is missing critical features modern text editors have, like an easy and powerful search feature. I hope this chapter will help you to make the transition to Vim easier.

You also just saw Vim's extensibility in action - the ability to extend search functionality with a plugin and an external program. In the future, keep in mind of what other features you wish to extend in Vim. Chances are, someone has created a plugin or there is a program for it already. Next, you'll learn about a very important topic in Vim: Vim grammar.

Ch04. Vim Grammar

It is easy to get intimidated by the complexity of Vim commands. If you see a Vim user doing `gUfV` or `1GdG`, you may not immediately know what these commands do. In this chapter, I will break down the general structure of Vim commands into a simple grammar rule.

This is the most important chapter in the entire guide. Once you understand the underlying grammatical structure, you will be able to "speak" to Vim. By the way, when I say *Vim language* in this chapter, I am not talking about Vimscript language (Vim's built-in programming language, you will learn that in later chapters).

How To Learn A Language

I am not a native English speaker. I learned English when I was 13 when I moved to the US. There are three things you need to do to learn to speak a new language:

1. Learn grammar rules.
2. Increase vocabulary.
3. Practice, practice, practice.

Likewise, to speak Vim language, you need to learn the grammar rules, increase vocabulary, and practice until you can run the commands without thinking.

Grammar Rule

There is only one grammar rule in Vim language:

```
verb + noun
```

That's it!

This is like saying these English phrases:

- *"Eat (verb) a donut (noun)"*
- *"Kick (verb) a ball (noun)"*
- *"Learn (verb) the Vim editor (noun)"*

Now you need to build up your vocabulary with basic Vim verbs and nouns.

Nouns (Motions)

Nouns are Vim motions. Motions are used to move around in Vim. Below is a list of some of Vim motions:

```
h    Left
j    Down
k    Up
l    Right
w    Move forward to the beginning of the next word
}    Jump to the next paragraph
$    Go to the end of the line
```

You will learn more about motions in the next chapter, so don't worry too much if you don't understand some of them.

Verbs (Operators)

According to `:h operator`, Vim has 16 operators. However, in my experience, learning these 3 operators is enough for 80% of my editing needs:

```
y    Yank text (copy)
d    Delete text and save to register
c    Delete text, save to register, and start insert mode
```

Btw, after you yank a text, you can paste it with `p` (after the cursor) or `P` (before the cursor).

Verb And Noun

Now that you know basic nouns and verbs, let's apply the grammar rule, verb + noun! Suppose you have this expression:

```
const learn = "vim";
```

- To yank everything from your current location to the end of the line: `y$`.
- To delete from your current location to the beginning of the next word: `dw`.
- To change from your current location to the end of the current paragraph, say `c}`.

Motions also accept count number as arguments (I will discourse this in the next chapter). If you need to go up 3 lines, instead of pressing `k` 3 times, you can do `3k`. Count works with Vim grammar.

- To yank two characters to the left: `y2h`.
- To delete the next two words: `d2w`.
- To change the next two lines: `c2j`.

Right now, you may have to think long and hard to do even a simple command. You're not alone. When I first started, I had similar struggles but I got faster in time. So will you. Repetition, repetition, repetition.

As a side note, linewise operations (operations affecting the entire line) are common operations in text editing. In general, by typing an operator command twice, Vim performs a linewise operation for that action. For example, `dd`, `yy`, and `cc` perform **deletion**, **yank**, and **change** on the entire line. Try this with other operators!

This is really cool. I am seeing a pattern here. But I am not quite done yet. Vim has one more type of noun: text objects.

More Nouns (Text Objects)

Imagine you are somewhere inside a pair of parentheses like `(hello Vim)` and you need to delete the entire phrase inside the parentheses. How can you quickly do it? Is there a way to delete the "group" you are inside of?

The answer is yes. Texts often come structured. They often contain parentheses, quotes, brackets, braces, and more. Vim has a way to capture this structure with text objects.

Text objects are used with operators. There are two types of text objects: inner and outer text objects.

```
i + object    Inner text object
a + object    Outer text object
```

Inner text object selects the object inside *without* the white space or the surrounding objects. Outer text object selects the object inside *including* the white space or the surrounding objects. Generally, an outer text object always selects more text than an inner text object. If your cursor is somewhere inside the parentheses in the expression `(hello vim)`:

- To delete the text inside the parentheses without deleting the parentheses: `di(`.
- To delete the parentheses and the text inside: `da(`.

Let's look at a different example. Suppose you have this Javascript function and your cursor is on the "H" in "Hello":

```
const hello = function() {
  console.log("Hello Vim");
  return true;
}
```

- To delete the entire "Hello Vim": `di(`.
- To delete the content of function (surrounded by `{}`): `di{`.
- To delete the "Hello" string: `diw`.

Text objects are powerful because you can target different objects from one location. You can delete the objects inside the parentheses, the function block, or the current word. Mnemonically, when you see `di(`, `di{`, and `diw`, you get a pretty good idea which text objects they represent: a pair of parentheses, a pair of braces, and a word.

Let's look at one last example. Suppose you have these HTML tags:

```
<div>
  <h1>Header1</h1>
  <p>Paragraph1</p>
  <p>Paragraph2</p>
</div>
```

If your cursor is on "Header1" text:

- To delete "Header1": `dit`.
- To delete `<h1>Header1</h1>`: `dat`.

If your cursor is on "div":

- To delete `h1` and both `p` lines: `dit`.
- To delete everything: `dat`.
- To delete "div": `di<`.

Below is a list of common text objects:

```
w         A word
p         A paragraph
s         A sentence
( or )    A pair of ( )
{ or }    A pair of { }
[ or ]    A pair of [ ]
< or >    A pair of < >
t         XML tags
"         A pair of " "
'         A Pair of ' '
`         A pair of ` `
```

To learn more, check out `:h text-objects`.

Composability And Grammar

Vim grammar is subset of Vim's composability feature. Let's discuss composability in Vim and why this is a great feature to have in a text editor.

Composability means having a set of general commands that can be combined (composed) to perform more complex commands. Just like in programming where you can create more complex abstractions from simpler abstractions, in Vim you can execute complex commands from simpler commands. Vim grammar is the manifestation of Vim's composable nature.

The true power of Vim's composability shines when it integrates with external programs. Vim has a filter operator (`!`) to use external programs as filters for our texts. Suppose you have this messy text below and you want to tabularize it:

```
Id|Name|Cuteness
01|Puppy|Very
02|Kitten|Ok
03|Bunny|Ok
```

This cannot be easily done with Vim commands, but you can get it done quickly with `column` terminal command (assuming your terminal has `column` command). With your cursor on "Id", run `!}column -t -s "|"`. Voila! Now you have this pretty tabular data with just one quick command.

```
Id  Name    Cuteness
01  Puppy   Very
02  Kitten  Ok
03  Bunny   Ok
```

Let's break down the command. The verb was `!` (filter operator) and the noun was `}` (go to next paragraph). The filter operator `!` accepted another argument, a terminal command, so I gave it `column -t -s "|"`. I won't go through how `column` worked, but in effect, it tabularized the text.

Suppose you want to not only tabularize your text, but to display only the rows with "Ok". You know that `awk` can do the job easily. You can do this instead:

```
!}column -t -s "|" | awk 'NR > 1 && /Ok/ {print $0}'
```

Result:

```
02  Kitten  Ok
03  Bunny   Ok
```

Great! The external command operator can also use pipe (`|`).

This is the power of Vim's composability. The more you know your operators, motions, and terminal commands, your ability to compose complex actions is *multiplied*.

Suppose you only know four motions, `w, $, }, G` and only one operator, `d`. You can do 8 actions: *move* 4 different ways (`w, $, }, G`) and *delete* 4 different targets (`dw, d$, d}, dG`). Then one day you learn about the uppercase (`gU`) operator. You have added not just one new ability to your Vim tool belt, but *four*: `gUw, gU$, gU}, gUG`. This makes at 12 tools in your Vim tool belt. Each new knowledge is a multiplier to your current abilities. If you know 10 motions and 5 operators, you have 60 moves (50 operations + 10 motions) in your arsenal. Vim has a line-number motion (`nG`) that gives you `n` motions, where `n` is how many lines you have in your file (to go to line 5, run `5G`). The search motion (`/`) practically gives you near unlimited number motions because you can search for anything. External command operator (`!`) gives you as many filtering tools as the number of terminal commands you know. Using a composable tool like Vim, everything you know can be linked together to do operations with increasing complexity. The more you know, the more powerful you become.

This composable behavior echoes Unix philosophy: *do one thing well*. An operator has one job: do Y. A motion has one job: go to X. By combining an operator with a motion, you predictably get YX: do Y on X.

Motions and operators are extendable. You can create custom motions and operators to add to your Vim toolbelt. The `vim-textobj-user`[19] plugin allows you to create your own text objects. It also contains a list[20] of user-made custom text objects.

Learn Vim Grammar The Smart Way

You just learned about Vim grammar's rule: `verb + noun`. One of my biggest Vim "AHA!" moments was when I had just learned about the uppercase (`gU`) operator and wanted to uppercase the current

[19] https://github.com/kana/vim-textobj-user
[20] https://github.com/kana/vim-textobj-user/wiki

word, I *instinctively* ran `gUiw` and it worked! The word was uppercased. At that moment, I finally began to understand Vim. My hope is that you will have your own "AHA!" moment soon, if not already.

The goal is this chapter is to show you the `verb + noun` pattern in Vim so you will approach learning Vim like learning a new language instead of memorizing every command combinations.

Learn the pattern and understand the implications. That's the smart way to learn.

Ch05. Moving In A File

In the beginning, moving with a keyboard feels slow and awkward but don't give up! Once you get used to it, you can go anywhere in a file faster than using a mouse.

In this chapter, you will learn the essential motions and how to use them efficiently. Keep in mind that this is **not** the entire motion that Vim has. The goal here is to introduce useful motions to become productive quickly. If you need to learn more, check out `:h motion.txt`.

Character Navigation

The most basic motion unit is moving one character left, down, up, and right.

```
h   Left
j   Down
k   Up
l   Right
```

You can also move with directional arrows. If you are just starting, feel free to use any method you're most comfortable with.

I prefer `hjkl` because my right hand can stay in the home row. Doing this gives me shorter reach to surrounding keys. To get used to `hjkl`, I actually disabled the arrow buttons when starting out by adding these in `~/.vimrc`:

```
noremap <Up> <NOP>
noremap <Down> <NOP>
noremap <Left> <NOP>
noremap <Right> <NOP>
```

There are also plugins to help break this bad habit. One of them is vim-hardtime[21]. To my surprise, it took me less than a week to get used to `hjkl`.

If you wonder why Vim uses `hjkl` to move, this is because Lear-Siegler ADM-3A terminal where Bill Joy wrote Vi, didn't have arrow keys and used `hjkl` as left/down/up/right.*

Relative Numbering

I think it is helpful to have `number` and `relativenumber` set. You can do it by having this on `.vimrc`:

[21] https://github.com/takac/vim-hardtime

```
set relativenumber number
```

This displays my current line number and relative line numbers.

It is easy why having a number on the left column is useful, but some of you may ask how having relative numbers on the left column may be useful. Having a relative number allows me to quickly see how many lines apart my cursor is from the target text. With this, I can easily spot that my target text is 12 lines below me so I can do `d12j` to delete them. Otherwise, if I'm on line 69 and my target is on line 81, I have to do mental calculation (81 - 69 = 12). Doing math while editing takes too much mental resources. The less I have to think about where I need to go, the better.

This is 100% personal preference. Experiment with `relativenumber` / `norelativenumber`, `number` / `nonumber` and use whatever you find most useful!

Count Your Move

Let's talk about the "count" argument. Vim motions accept a preceding numerical argument. I mentioned above that you can go down 12 lines with `12j`. The 12 in `12j` is the count number.

The syntax to use count with your motion is:

```
[count] + motion
```

You can apply this to all motions. If you want to move 9 characters to the right, instead of pressing `l` 9 times, you can do `9l`.

Word Navigation

Let's move to a larger motion unit: *word*. You can move to the beginning of the next word (`w`), to the end of the next word (`e`), to the beginning of the previous word (`b`), and to the end of the previous word (`ge`).

In addition, there is *WORD*, distinct from word. You can move to the beginning of the next WORD (`W`), to the end of the next WORD (`E`), to the beginning of the previous WORD (`B`), and to the end of the previous WORD (`gE`). To make it easy to remember, WORD uses the same letters as word, only uppercased.

```
w     Move forward to the beginning of the next word
W     Move forward to the beginning of the next WORD
e     Move forward one word to the end of the next word
E     Move forward one word to the end of the next WORD
b     Move backward to beginning of the previous word
B     Move backward to beginning of the previous WORD
ge    Move backward to end of the previous word
gE    Move backward to end of the previous WORD
```

So what are the similarities and differences between a word and a WORD? Both word and WORD are separated by non-blank characters. A word is a sequence of characters containing *only* `a-zA-Z0-9_`. A WORD is a sequence of all characters except white space (a white space means either space, tab, and EOL). To learn more, check out `:h word` and `:h WORD`.

For example, suppose you have:

```
const hello = "world";
```

With your cursor at the start of the line, to go to the end of the line with `1`, it will take you 21 key presses. Using `w`, it will take 6. Using `W`, it will only take 4. Both word and WORD are good options to travel short distance.

However, you can get from "c" to ";" in one keystroke with current line navigation.

Current Line Navigation

When editing, you often need to navigate horizontally in a line. To jump to the first character in current line, use `0`. To go to the last character in the current line, use `$`. Additionally, you can use `^` to go to the first non-blank character in the current line and `g_` to go to the last non-blank character in the current line. If you want to go to the column `n` in the current line, you can use `n|`.

```
0     Go to the first character in the current line
^     Go to the first nonblank char in the current line
g_    Go to the last non-blank char in the current line
$     Go to the last char in the current line
n|    Go the column n in the current line
```

You can do current line search with `f` and `t`. The difference between `f` and `t` is that `f` takes you to the first letter of the match and `t` takes you till (right before) the first letter of the match. So if you want to search for and land on "h", use `fh`. If you want to search for first "h" and land right before the match, use `th`. If you want to go to the *next* occurrence of the last current line search, use `;`. To go to the previous occurrence of the last current line match, use `,`.

`F` and `T` are the backward counterparts of `f` and `t`. To search backwards for "h", run `Fh`. To keep searching for "h" in the same direction, use `;`. Note that `;` after a `Fh` searches backward and `,` after `fh` searches forward.

```
f    Search forward for a match in the same line
F    Search backward for a match in the same line
t    Search forward for a match in the same line, stopping before match
T    Search backward for a match in the same line, stopping before match
;    Repeat the last search in the same line using the same direction
,    Repeat the last search in the same line using the opposite direction
```

Back at the previous example:

```
const hello = "world";
```

With your cursor at the start of the line, you can go to the last character in current line (";") with one keypress: `$`. If you want to go to "w" in "world", you can use `fw`. A good tip to go anywhere in a line is to look for least-common-letters like "j", "x", "z" near your target.

Sentence And Paragraph Navigation

Next two navigation units are sentence and paragraph.

Let's talk about what a sentence is first. A sentence ends with either `. ! ?` followed by an EOL, a space, or a tab. You can jump to the next sentence with `)` and the previous sentence with `(`.

```
(    Jump to the previous sentence
)    Jump to the next sentence
```

Let's look at some examples. Which phrases do you think are sentences and which aren't? Try navigating with `(` and `)` in Vim!

```
I am a sentence. I am another sentence because I end with a period. I am still a sen\
tence when ending with an exclamation point! What about question mark? I am not quit\
e a sentence because of the hyphen - and neither semicolon ; nor colon :

There is an empty line above me.
```

By the way, if you're having a problem with Vim not counting a sentence for phrases separated by `.` followed by a single line, you might be in `'compatible'` mode. Add `set nocompatible` into vimrc. In Vi, a sentence is a `.` followed by **two** spaces. You should have `nocompatible` set at all times.

Let's talk what a paragraph is. A paragraph begins after each empty line and also at each set of a paragraph macro specified by the pairs of characters in paragraphs option.

```
{    Jump to the previous paragraph
}    Jump to the next paragraph
```

If you're not sure what a paragraph macro is, do not worry. The important thing is that a paragraph begins and ends after an empty line. This should be true most of the time.

Let's look at this example. Try navigating around with `}` and `{` (also, play around with sentence navigations `( )` to move around too!)

```
Hello. How are you? I am great, thanks!
Vim is awesome.
It may not easy to learn it at first...- but we are in this together. Good luck!

Hello again.

Try to move around with ), (, }, and {. Feel how they work.
You got this.
```

Check out `:h sentence` and `:h paragraph` to learn more.

Match Navigation

Programmers write and edit codes. Codes typically use parentheses, braces, and brackets. You can easily get lost in them. If you're inside one, you can jump to the other pair (if it exists) with `%`. You can also use this to find out whether you have matching parentheses, braces, and brackets.

```
%    Navigate to another match, usually works for (), [], {}
```

Let's look at a Scheme code example because it uses parentheses extensively. Move around with `%` inside different parentheses.

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else
          (+ (fib (- n 1)) (fib (- n 2)))
        )))
```

I personally like to complement `%` with visual indicators plugins like vim-rainbow[22]. For more, check out `:h %`.

[22] https://github.com/frazrepo/vim-rainbow

Line Number Navigation

You can jump to line number `n` with `nG`. For example, if you want to jump to line `7`, use `7G`. To jump to the first line, use either `1G` or `gg`. To jump to the last line, use `G`.

Often you don't know exactly what line number your target is, but you know it's approximately at `70%` of the whole file. In this case, you can do `70%`. To jump halfway through the file, you can do `50%`.

```
gg    Go to the first line
G     Go to the last line
nG    Go to line n
n%    Go to n% in file
```

By the way, if you want to see total lines in a file, you can use `Ctrl-g`.

Window Navigation

To quickly go to the top, middle, or bottom of your *window*, you can use `H`, `M`, and `L`.

You can also pass a count to `H` and `L`. If you use `10H`, you will go to `10` lines below the top of window. If you use `3L`, you will go to `3` lines above the last line of window.

```
H     Go to top of screen
M     Go to medium screen
L     Go to bottom of screen
nH    Go n line from top
nL    Go n line from bottom
```

Scrolling

To scroll, you have `3` speed increments: full-screen (`Ctrl-F/Ctrl-B`), half-screen (`Ctrl-D/Ctrl-U`), and line (`Ctrl-E/Ctrl-Y`).

```
Ctrl-E    Scroll down a line
Ctrl-D    Scroll down half screen
Ctrl-F    Scroll down whole screen
Ctrl-Y    Scroll up a line
Ctrl-U    Scroll up half screen
Ctrl-B    Scroll up whole screen
```

You can also scroll relatively to the current line (zoom screen sight):

```
zt    Bring the current line near the top of your screen
zz    Bring the current line to the middle of your screen
zb    Bring the current line near the bottom of your screen
```

Search Navigation

Often you know that a phrase exists inside a file. You can use search navigation to very quickly reach your target. To search for a phrase, you can use `/` to search forward and `?` to search backward. To repeat the last search you can use `n`. To repeat the last search going opposite direction, you can use `N`.

```
/    Search forward for a match
?    Search backward for a match
n    Repeat last search (same direction of previous search)
N    Repeat last search (opposite direction of previous search)
```

Suppose you have this text:

```
let one = 1;
let two = 2;
one = "01";
one = "one";
let onetwo = 12;
```

If you are searching for "let", run `/let`. To quickly search for "let" again, you can just do `n`. To search for "let" again in opposite direction, run `N`. If you run `?let`, it will search for "let" backwards. If you use `n`, it will now search for "let" backwards (`N` will search for "let" forwards now).

You can enable search highlight with `set hlsearch`. Now when you search for `/let`, it will highlight *all* matching phrases in the file. In addition, you can set incremental search with `set incsearch`. This will highlight the pattern while typing. By default, your matching phrases will remain highlighted until you search for another phrase. This can quickly turn into an annoyance. To disable highlight, you can run `:nohlsearch`. Because I use this no-highlight feature frequently, I created a map in vimrc:

```
nnoremap <esc><esc> :noh<return><esc>
```

You can quickly search for the text under the cursor with `*` to search forward and `#` to search backward. If your cursor is on the string "one", pressing `*` will be the same as if you had done `/\<one\>`.

Both `\<` and `\>` in `/\<one\>` mean whole word search. It does not match "one" if it is a part of a bigger word. It will match for the word "one" but not "onetwo". If your cursor is over "one" and you want to search forward to match whole or partial words like "one" and "onetwo", you need to use `g*` instead of `*`.

```
*     Search for whole word under cursor forward
#     Search for whole word under cursor backward
g*    Search for word under cursor forward
g#    Search for word under cursor backward
```

Marking Position

You can use marks to save your current position and return to this position later. It's like a bookmark for text editing. You can set a mark with `mx`, where `x` can be any alphabetical letter `a-zA-Z`. There are two ways to return to mark: exact (line and column) with `` `x `` and linewise (`'x`).

```
ma    Mark position with mark "a"
`a    Jump to line and column "a"
'a    Jump to line "a"
```

There is a difference between marking with lowercase letters (a-z) and uppercase letters (A-Z). Lowercase alphabets are local marks and uppercase alphabets are global marks (sometimes known as file marks).

Let's talk about local marks. Each buffer can have its own set of local marks. If I have two files opened, I can set a mark "a" (`ma`) in the first file and another mark "a" (`ma`) in the second file.

Unlike local marks where you can have a set of marks in each buffer, you only get one set of global marks. If you set `mA` inside `myFile.txt`, the next time you run `mA` in a different file, it will overwrite the first "A" mark. One advantage of global marks is you can jump to any global mark even if you are inside a completely different project. Global marks can travel across files.

To view all marks, use `:marks`. You may notice from the marks list there are more marks other than `a-zA-Z`. Some of them are:

```
''    Jump back to the last line in current buffer before jump
``    Jump back to the last position in current buffer before jump
`[    Jump to beginning of previously changed / yanked text
`]    Jump to the ending of previously changed / yanked text
`<    Jump to the beginning of last visual selection
`>    Jump to the ending of last visual selection
`0    Jump back to the last edited file when exiting vim
```

There are more marks than the ones listed above. I won't cover them here because I think they are rarely used, but if you're curious, check out `:h marks`.

Jump

In Vim, you can "jump" to a different file or different part of a file with some motions. Not all motions count as a jump, though. Going down with `j` does not count as a jump. Going to line 10 with `10G` counts as a jump.

Here are the commands Vim consider as “jump” commands:

```
'       Go to the marked line
`       Go to the marked position
G       Go to the line
/       Search forward
?       Search backward
n       Repeat the last search, same direction
N       Repeat the last search, opposite direction
%       Find match
(       Go to the last sentence
)       Go to the next sentence
{       Go to the last paragraph
}       Go to the next paragraph
L       Go to the the last line of displayed window
M       Go to the middle line of displayed window
H       Go to the top line of displayed window
[[      Go to the previous section
]]      Go to the next section
:s      Substitute
:tag    Jump to tag definition
```

I don’t recommend memorizing this list. A good rule of thumb is, any motion that moves farther than a word and current line navigation is probably a jump. Vim keeps track of where you’ve been when you move around and you can see this list inside `:jumps`.

For more, check out `:h jump-motions`.

Why are jumps useful? Because you can navigate the jump list with `Ctrl-O` to move up the jump list and `Ctrl-I` to move down the jump list. You can jump across different files, which I will discuss more in the next part.

Learn Navigation The Smart Way

If you are new to Vim, this is a lot to learn. I do not expect anyone to remember everything immediately. It takes time before you can execute them without thinking.

I think the best way to get started is to memorize a few essential motions. I recommend starting out with these 10 motions: `h, j, k, l, w, b, G, /, ?, n`. Repeat them sufficiantly until you can use them without thinking.

To improve your navigation skill, here are my suggestions:

1. Watch for repeated actions. If you find yourself doing `l` repeatedly, look for a motion that will take you forward faster. You will find that you can use `w`. If you catch yourself repeatedly doing `w`, look if there is a motion that will take you across the current line quickly. You will find that you can use the `f`. If you can describe your need succintly, there is a good chance Vim has a way to do it.
2. Whenever you learn a new move, spend some time until you can do it without thinking.

Finally, realize that you do not need to know every single Vim command to be productive. Most Vim users don't. I don't. Learn the commands that will help you accomplish your task at that moment.

Take your time. Navigation skill is a very important skill in Vim. Learn one small thing every day and learn it well.

Ch06. Insert Mode

Insert mode is the default mode of many text editors. In this mode, what you type is what you get.

However, that does not mean there isn't much to learn. Vim's insert mode contains many useful features. In this chapter, you will learn how to use these insert mode features in Vim to improve your typing efficiency.

Ways To Go To Insert Mode

There are many ways to get into insert mode from the normal mode. Here are some of them:

```
i    Insert text before the cursor
I    Insert text before the first non-blank character of the line
a    Append text after the cursor
A    Append text at the end of line
o    Starts a new line below the cursor and insert text
O    Starts a new line above the cursor and insert text
s    Delete the character under the cursor and insert text
S    Delete the current line and insert text
gi   Insert text in same position where the last insert mode was stopped
gI   Insert text at the start of line (column 1)
```

Notice the lowercase / uppercase pattern. For each lowercase command, there is an uppercase counterpart. If you are new, don't worry if you don't remember the whole list above. Start with `i` and `o`. They should be enough to get you started. Gradually learn more over time.

Different Ways To Exit Insert Mode

There are a few different ways to return to the normal mode while in the insert mode:

```
<Esc>     Exits insert mode and go to normal mode
Ctrl-[    Exits insert mode and go to normal mode
Ctrl-C    Like Ctrl-[ and <Esc>, but does not check for abbreviation
```

I find `<Esc>` key too far to reach, so I map my computer `<Caps-Lock>` to behave like `<Esc>`. If you search for Bill Joy's ADM-3A keyboard (Vi creator), you will see that the `<Esc>` key is not located

on far top left like modern keyboards, but to the left of `q` key. This is why I think it makes sense to map `<Caps lock>` to `<Esc>`.

Another common convention I have seen Vim users do is mapping `<Esc>` to `jj` or `jk` in insert mode. If you prefer this option add this one of those lines (or both) in your vimrc file.

```
inoremap jj <Esc>
inoremap jk <Esc>
```

Repeating Insert Mode

You can pass a count parameter before entering insert mode. For example:

```
10i
```

If you type "hello world!" and exit insert mode, Vim will repeat the text `10` times. This will work with any insert mode method (ex: `10I`, `11a`, `12o`).

Deleting Chunks In Insert Mode

When you make a typing mistake, it can be cumbersome to type `<Backspace>` repeatedly. It may make more sense to go to normal mode and delete your mistake. You can also delete several characters at a time while in insert mode.

```
Ctrl-H    Delete one character
Ctrl-W    Delete one word
Ctrl-U    Delete the entire line
```

Insert From Register

Vim registers can store texts for future use. To insert a text from any named register while in insert mode, type `Ctrl-R` plus the register symbol. There are many symbols you can use, but for this section, let's cover only the named registers (a-z).

To see it in action, first you need to yank a word to register a. Move your cursor on any word. Then type:

```
"ayiw
```

- `"a` tells Vim that the target of your next action will go to register a.
- `yiw` yanks inner word. Review the chapter on Vim grammar for a refresher.

Register a now contains the word you just yanked. While in insert mode, to paste the text stored in register a:

```
Ctrl-R a
```

There are multiple types of registers in Vim. I will cover them in greater detail in a later chapter.

Scrolling

Did you know that you can scroll while inside insert mode? While in insert mode, if you go to `Ctrl-X` sub-mode, you can do additional operations. Scrolling is one of them.

```
Ctrl-X Ctrl-Y    Scroll up
Ctrl-X Ctrl-E    Scroll down
```

Autocompletion

As mentioned above, if you press `Ctrl-X` from insert mode, Vim will enter a sub-mode. You can do text autocompletion while in this insert mode sub-mode. Although it is not as good as intellisense[23] or any other Language Server Protocol (LSP), but for something that is available right out of the box, it is a very capable feature.

Here are some useful autocomplete commands to get started:

```
Ctrl-X Ctrl-L           Insert a whole line
Ctrl-X Ctrl-N           Insert a text from current file
Ctrl-X Ctrl-I           Insert a text from included files
Ctrl-X Ctrl-F           Insert a file name
```

When you trigger autocompletion, Vim will display a pop-up window. To navigate up and down the pop-up window, use `Ctrl-N` and `Ctrl-P`.

Vim also has two autocompletion shortcuts that don't involve the `Ctrl-X` sub-mode:

```
Ctrl-N            Find the next word match
Ctrl-P            Find the previous word match
```

In general, Vim looks at the text in all available buffers for autocompletion source. If you have an open buffer with a line that says "Chocolate donuts are the best":

- When you type "Choco" and do `Ctrl-X Ctrl-L`, it will match and print the entire line.
- When you type "Choco" and do `Ctrl-P`, it will match and print the word "Chocolate".

Autocomplete is a vast topic in Vim. This is just the tip of the iceberg. To learn more, check out `:h ins-completion`.

[23] https://code.visualstudio.com/docs/editor/intellisense

Executing A Normal Mode Command

Did you know Vim can execute a normal mode command while in insert mode?

While in insert mode, if you press `Ctrl-O`, you'll be in insert-normal sub-mode. If you look at the mode indicator on bottom left, normally you will see `-- INSERT --`, but pressing `Ctrl-O` changes it to `-- (insert) --`. In this mode, you can do *one* normal mode command. Some things you can do:

Centering and jumping

```
Ctrl-O zz       Center window
Ctrl-O H/M/L    Jump to top/middle/bottom window
Ctrl-O 'a       Jump to mark a
```

Repeating text

```
Ctrl-O 100ihello    Insert "hello" 100 times
```

Executing terminal commands

```
Ctrl-O !! curl https://google.com    Run curl
Ctrl-O !! pwd                        Run pwd
```

Deleting faster

```
Ctrl-O dtz    Delete from current location till the letter "z"
Ctrl-O D      Delete from current location to the end of the line
```

Learn Insert Mode The Smart Way

If you are like me and you come from another text editor, it can be tempting to stay in insert mode. However, staying in insert mode when you're not entering a text is an anti-pattern. Develop a habit to go to normal mode when your fingers aren't typing new texts.

When you need to insert a text, first ask yourself if that text already exists. If it does, try to yank or move that text instead of typing it. If you have to use insert mode, see if you can autocomplete that text whenever possible. Avoid typing the same word more than once if you can.

Ch07. The Dot Command

In general, you should try to avoid redoing what you just did whenever possible. In this chapter, you will learn how to use the dot command to easily redo the previous change. It is a versatile command for reducing simple repetitions.

Usage

Just like its name, you can use the dot command by pressing the dot key (`.`).

For example, if you want to replace all "let" with "const" in the following expressions:

```
let one = "1";
let two = "2";
let three = "3";
```

- Search with `/let` to go to the match.
- Change with `cwconst<Esc>` to replace "let" with "const".
- Navigate with `n` to find the next match using the previous search.
- Repeat what you just did with the dot command (`.`).
- Continue pressing `n . n .` until you replace every word.

Here the dot command repeated the `cwconst<Esc>` sequence. It saved you from typing eight keystrokes in exchange for just one.

What Is A Change?

If you look at the definition of the dot command (`:h .`), it says that the dot command repeats the last change. What is a change?

Any time you update (add, modify, or delete) the content of the current buffer, you are making a change. The exceptions are updates done by command-line commands (the commands starting with `:`) do not count as a change.

In the first example, `cwconst<Esc>` was the change. Now suppose you have this text:

```
pancake, potatoes, fruit-juice,
```

To delete the text from the start of the line to the next occurrence of a comma, first delete to the comma, then repeat twice it with `df,` . . .

Let's try another example:

```
pancake, potatoes, fruit-juice,
```

This time, your task is to delete the comma, not the breakfast items. Go to the first comma using, delete it, then repeat two more times with `f,x` . . Easy, right? Wait a minute, it didn't work! Why?

A change excludes motions because it does not update buffer content. The command `f,x` consisted of two actions: the command `f,` to move the cursor to "," and `x` to delete a character. Only the latter, `x`, caused a change. Contrast that with `df,` from the earlier example. In it, `f,` is a directive to the delete operator `d`, not a motion to move the cursor. The `f,` in `df,` and `f,x` have two very different roles.

Let's finish the last task. After you run `f,` then `x`, go to the next comma with `;` to repeat the latest `f`. Finally, use `.` to delete the character under the cursor. Repeat `; . ; .` until everything is deleted. The full command is `f,x;.;.`.

Let's try another one:

```
pancake
potatoes
fruit-juice
```

Lets add a comma at the end of each line. Starting at the first line, do `A,<Esc>j`. By now, you realize that `j` does not cause a change. The change here is only `A,`. You can move and repeat the change with `j . j .`. The full command is `A,<Esc>j.j.`.

Every action from the moment you press the insert command operator (`A`) until you exit the insert command (`<Esc>`) is considered as a change.

Multi-line Repeat

Suppose you have this text:

```
let one = "1";
let two = "2";
let three = "3";
const foo = "bar';
let four = "4";
let five = "5";
let six = "6";
let seven = "7";
let eight = "8";
let nine = "9";
```

Your goal is to delete all lines except the "foo" line. First, delete the first three lines with `d2j`, then to the line below the "foo" line. On the next line, use the dot command twice. The full command is `d2jj...`

Here the change was `d2j`. In this context, `2j` was not a motion, but a part of the delete operator.

Let's look at another example:

```
zlet zzone = "1";
zlet zztwo = "2";
zlet zzthree = "3";
let four = "4";
```

Let's remove all the z's. Starting from the first character on the first line, visually select the only the first z from the first three lines with blockwise visual mode (`Ctrl-Vjj`). If you're not familiar with blockwise visual mode, I will cover them in a later chapter. Once you have the three z's visually selected, delete them with the delete operator (`d`). Then move to the next word (`w`) to the next z. Repeat the change two more times (`..`). The full command is `Ctrl-vjjdw...`

When you deleted a column of three z's (`Ctrl-vjjd`), it was counted as a change. Visual mode operation can be used to target multiple lines as part of a change.

Including A Motion In A Change

Let's revisit the first example in this chapter. Recall that the command `/letcwconst<Esc>` followed by `n . n .` replaced all "let" with "const" in the following expressions:

```
let one = "1";
let two = "2";
let three = "3";
```

There is a faster way to accomplish this. After you searched `/let`, run `cgnconst<Esc>` then `. ..`

`gn` is a motion that searches forward for the last search pattern (in this case, `/let`) and automatically does a visual highlight. To replace the next occurrence, you no longer have to move and repeat the change (`n . n .`), but only repeat (`. .`). You do not have to use search motions anymore because searching the next match is now part of the change!

When you are editing, always be on the lookout for motions that can do several things at once like `gn` whenever possible.

Learn The Dot Command The Smart Way

The dot command's power comes from exchanging several keystrokes for one. It is probably not a profitable exchange to use the dot command for single key operations like `x`. If your last change requires a complex operation like `cgnconst<Esc>`, the dot command reduces nine keypresses into one, a very profitable trade-off.

When editing, think about repeatability. For example, if I need to remove the next three words, is it more economical to use `d3w` or to do `dw` then `.` two times? Will you be deleting a word again? If so, then it makes sense to use `dw` and repeat it several times instead of `d3w` because `dw` is more reusable than `d3w`.

The dot command is the simpversatile command for automating single changes. In a later chapter, you will learn how to automate more complex actions with Vim macros. But first, let's learn about registers to store and retrieve text.

Ch08. Registers

Learning Vim registers is like learning algebra for the first time. You didn't think you need it until you needed it.

You've probably used Vim registers when you yanked or deleted a text then pasted it with `p` or `P`. However, did you know that Vim has 10 different types of registers? Used correctly, Vim registers can save you from repetitive typing.

In this chapter, I will go over all Vim register types and how to use them efficiently.

The Ten Register Types

Here are the 10 Vim register types:

1. The unnamed register (`""`).
2. The numbered registers (`"0-9`).
3. The small delete register (`"-`).
4. The named registers (`"a-z`).
5. The read-only registers (`":`, `".`,and `"%`).
6. The alternate buffer register (`"#`).
7. The expression register (`"=`).
8. The selection registers (`"*` and `"+`).
9. The black hole register (`"_`).
10. The last search pattern register (`"/`).

Register Operators

To use registers, you need to first store them with operators. Here are some operators that store values to registers:

```
y    Yank (copy)
c    Delete text and start insert mode
d    Delete text
```

There are more operators (like `s` or `x`), but the above are the useful ones. The rule of thumb is, if an operator can remove a text, it probably stores the text to registers.

To paste a text from registers, you can use:

```
p    Paste the text after the cursor
P    Paste the text before the cursor
```

Both `p` and `P` accept a count and a register symbol as arguments. For example, to paste ten times, do `10p`. To paste the text from register a, do `"ap`. To paste the text from register a ten times, do `10"ap`. By the way, the `p` actually technically stands for "put", not "paste", but I figure paste is a more conventional word.

The general syntax to get the content from a specific register is `"a`, where `a` is the register symbol.

Calling Registers From Insert Mode

Everything you learn in this chapter can also be executed in insert mode. To get the text from register a, normally you do `"ap`. But if you are in insert mode, run `Ctrl-R a`. The syntax to call registers from insert mode is:

```
Ctrl-R a
```

Where `a` is the register symbol. Now that you know how to store and retrieve registers, let's dive in!

The Unnamed Register

To get the text from the unnamed register, do `""p`. It stores the last text you yanked, changed, or deleted. If you do another yank, change, or delete, Vim will automatically replace the old text. The unnamed register is like a computer's standard copy / paste operation.

By default, `p` (or `P`) is connected to the unnamed register (from now on I will refer to the unnamed register with `p` instead of `""p`).

The Numbered Registers

Numbered registers automatically fill themselves up in ascending order. There are `2` different numbered registers: the yanked register (`0`) and the numbered registers (`1-9`). Let's discuss the yanked register first.

The Yanked Register

If you yank an entire line of text (`yy`), Vim actually saves that text in two registers:

1. The unnamed register (`p`).

2. The yanked register (`"0p`).

When you yank a different text, Vim will update both the yanked register and the unnamed register. Any other operations (like delete) will not be stored in register `0`. This can be used to your advantage, because unless you do another yank, the yanked text will always be there, no matter how many changes and deletions you do.

For example, if you:

1. Yank a line (`yy`)
2. Delete a line (`dd`)
3. Delete another line (`dd`)

The yanked register will have the text from step one.

If you:

1. Yank a line (`yy`)
2. Delete a line (`dd`)
3. Yank another line (`yy`)

The yanked register will have the text from step three.

One last tip, while in insert mode, you can quickly paste the text you just yanked using `Ctrl-R 0`.

The Non-zero Numbered Registers

When you change or delete a text that is at least one line long, that text will be stored in the numbered registers 1-9 sorted by the most recent.

For example, if you have these lines:

```
line three
line two
line one
```

With your cursor on “line three”, delete them one by one with `dd`. Once all lines are deleted, register 1 should contain “line one” (most recent), register two “line two” (second most recent), and register three “line three” (oldest. To get the content from register one, do `"1p`.

As a side note, these numbered registers are automatically incremented when using the dot command. If your numbered register one (`"1`) contains “line one”, register two (`"2`) “line two”, and register three (`"3`) “line three”, you can paste them sequentially with this trick:

- Do `"1P` to paste the content from the numbered register one (“1).
- Do `.` to paste the content from the numbered register two (“2).
- Do `.` to paste the content from the numbered register three (“3).

This trick works with any numbered register. If you started with `"5P`, `.` would do `"6P`, `.` again would do `"7P`, and so on.

Small deletions like a word deletion (`dw`) or word change (`cw`) do not get stored in the numbered registers. They are stored in the small delete register (`"-`), which I will discuss next.

The Small Delete Register

Changes or deletions less than one line are not stored in the numbered registers 0-9, but in the small delete register (`"-`).

For example:

1. Delete a word (`diw`)
2. Delete a line (`dd`)
3. Delete a line (`dd`)

`"-p` gives you the deleted word from step one.

Another example:

1. I delete a word (`diw`)
2. I delete a line (`dd`)
3. I delete a word (`diw`)

`"-p` gives you the deleted word from step three. `"1p` gives you the deleted line from step two. Unfortunately, there is no way to retrieve the deleted word from step one because the small delete register only stores one item. However, if you want to preserve the text from step one, you can do it with the named registers.

The Named Register

The named registers are Vim’s most versatile register. It can store yanked, changed, and deleted texts into registers a-z. Unlike the previous 3 register types you’ve seen which automatically stores texts into registers, you have to explicitly tell Vim to use the named register, giving you full control.

To yank a word into register a, you can do it with `"ayiw`.

- `"a` tells Vim that the next action (delete / change / yank) will be stored in register a.
- `yiw` yanks the word.

To get the text from register a, run `"ap`. You can use all twenty-six alphabetical characters to store twenty-six different texts with named registers.

Sometimes you may want to add to your existing named register. In this case, you can append your text instead of starting all over. To do that, you can use the uppercase version of that register. For example, suppose you have the word "Hello " already stored in register a. If you want to add "world" into register a, you can find the text "world" and yank it using A register (`"Ayiw`).

The Read-Only Registers

Vim has three read-only registers: `.`, `:`, and `%`. They are pretty simple to use:

```
.    Stores the last inserted text
:    Stores the last executed command-line
%    Stores the name of current file
```

If the last text you wrote was "Hello Vim", running `".p` will print out the text "Hello Vim". If you want to get the name of current file, run `"%p`. If you run `:s/foo/bar/g` command, running `":p` will print out the literal text "s/foo/bar/g".

The Alternate File Register

In Vim, `#` usually represents the alternate file. An alternative file is the last file you opened. To insert the name of the alternate file, you can use `"#p`.

The Expression Register

Vim has an expression register, `"=`, to evaluate expressions.

To evaluate mathematical expressions `1 + 1`, run:

```
"=1+1<Enter>p
```

Here, you are telling Vim that you are using the expression register with `"=`. Your expression is (`1 + 1`). You need to type `p` to get the result. As mentioned earlier, you can also access the register from insert mode. To evaluate mathematical expression from insert mode, you can do:

```
Ctrl-R =1+1
```

You can also get the values from any register via the expression register when appended with `@`. If you wish to get the text from register a:

```
"=@a
```

Then press `<Enter>`, then `p`. Similarly, to get values from register a while in insert mode:

```
Ctrl-r =@a
```

Expression is a vast topic in Vim, so I will only cover the basics here. I will address expressions in more details in later Vimscript chapters.

The Selection Registers

Don't you sometimes wish that you can copy a text from external programs and paste it locally in Vim, and vice versa? With Vim's selection registers, you can. Vim has two selection registers: `quotestar` (`"*`) and `quoteplus` (`"+`). You can use them to access copied text from external programs.

If you are on an external program (like Chrome browser) and you copy a block of text with `Ctrl-C` (or `Cmd-C`, depending on your OS), normally you wouldn't be able to use `p` to paste the text in Vim. However, both Vim's `"+` and `"*` are connected to your clipboard, so you can actually paste the text with `"+p` or `"*p`. Conversely, if you yank a word from Vim with `"+yiw` or `"*yiw`, you can paste that text in the external program with `Ctrl-V` (or `Cmd-V`). Note that this only works if your Vim program comes with the `+clipboard` option (to check it, run `:version`).

You may wonder if `"*` and `"+` do the same thing, why does Vim have two different registers? Some machines use X11 window system. This system has 3 types of selections: primary, secondary, and clipboard. If your machine uses X11, Vim uses X11's *primary* selection with the `quotestar` (`"*`) register and X11's *clipboard* selection with the `quoteplus` (`"+`) register. This is only applicable if you have `+xterm_clipboard` option available in your Vim build. If your Vim doesn't have `xterm_clipboard`, it's not a big deal. It just means that both `quotestar` and `quoteplus` are interchangeable (mine doesn't either).

I find doing `=*p` or `=+p` to be cumbersome. To make Vim to paste copied text from the external program with just `p`, you can add this in your vimrc:

```
set clipboard=unnamed
```

Now when I copy a text from an external program, I can paste it with the unnamed register, `p`. I can also copy a text from Vim and paste it to an external program. If you have `+xterm_clipboard` on, you may want to use both `unnamed` and `unnamedplus` clipboard options.

The Black Hole Register

Each time you delete or change a text, that text is stored in Vim register automatically. There will be times when you don't want to save anything into the register. How can you do that?

You can use the black hole register (`"_`). To delete a line and not have Vim store the deleted line into any register, use `"_dd`.

The black hole register is like the `/dev/null` of registers.

The Last Search Pattern Register

To paste your last search (`/` or `?`), you can use the last search pattern register (`"/`). To paste the last search term, use `"/p`.

Viewing The Registers

To view all your registers, use the `:register` command. To view only registers "a, "1, and "-, use `:register a 1 -`.

There is a plugin called vim-peekaboo[24] that lets you to peek into the contents of the registers when you hit `"` or `@` in normal mode and `Ctrl-R` in insert mode. I find this plugin very useful because most times, I can't remember the content in my registers. Give it a try!

Executing A Register

The named registers are not just for storing texts. They can also execute macros with `@`. I will go over macros in the next chapter.

Keep in mind since macros are stored inside Vim registers, you can accidentally overwrite the stored text with macros. If you store the text "Hello Vim" in register a and you later record a macro in the same register (`qa{macro-sequence}q`), that macro will overwrite your "Hello Vim" text stored earlier.

Clearing A Register

Technically, there is no need to clear any register because the next register you store under the same name will overwrite it. However, you can quickly clear any named register by recording an empty macro. For example, if you run `qaq`, Vim will record an empty macro in the register a.

Another alternative is to run the command `:call setreg('a', '')` where "a is the register a.

One more way to clear register is to set the content of "a register to an empty string with the expression `:let @a = ''`.

[24] https://github.com/junegunn/vim-peekaboo

Putting The Content Of A Register

You can use the `:put` command to paste the content of any one register. For example, if you run `:put a`, Vim will print the content of register a below the current line. This behaves much like `"ap`, with the difference that the normal mode command `p` prints the register content after the cursor and the command `:put` prints the register content at newline.

Since `:put` is a command-line command, you can pass it an address. `:10put a` will paste text from register a to below line 10.

One cool trick to pass `:put` with the black hole register (`"_`). Since the black hole register does not store any text, `:put _` will insert a blank line instead. You can combine this with the global command to insert multiple blank lines. For example, to insert blank lines below all lines that contain the text "end", run `:g/end/put _`. You will learn about the global command later.

Learning Registers The Smart Way

You made it to the end. Congratulations! If you are feeling overwhelmed by the sheer information, you are not alone. When I first started learning about Vim registers, there were way too much information to take at once.

I don't think you should memorize all the registers immediately. To become productive, you can start by using only these 3 registers:

1. The unnamed register (`""`).
2. The named registers (`"a-z`).
3. The numbered registers (`"0-9`).

Since the unnamed register defaults to `p` and `P`, you only have to learn two registers: the named registers and the numbered registers. Gradually learn more registers when you need them. Take your time.

The average human has a limited short-term memory capacity, about 5 - 7 items at once. That is why in my everyday editing, I only use about 5 - 7 named registers. There is no way I can remember all twenty-six in my head. I normally start with register a, then b, ascending the alphabetical order. Try it and experiment around to see what technique works best for you.

Vim registers are powerful. Used strategically, it can save you from typing countless repeating texts. Next, let's learn about macros.

Ch09. Macros

When editing files, you may find yourself repeating the same actions. Wouldn't it be nice if you can do those actions once and replay them whenever you need it? With Vim macros, you can record actions and store them inside Vim registers to be executed whenever you need it.

In this chapter, you will learn how to use macros to automate mundane tasks (plus it looks cool to watch your file edit itself).

Basic Macros

Here is the basic syntax of a Vim macro:

```
qa                    Start recording a macro in register a
q (while recording)   Stop recording macro
```

You can choose any lowercase letters (a-z) to store macros. Here is how you can execute a macro:

```
@a    Execute macro from register a
@@    Execute the last executed macros
```

Suppose you have this text and you want to uppercase everything on each line:

```
hello
vim
macros
are
awesome
```

With your cursor at the start of the line "hello", run:

```
qa0gU$jq
```

The breakdown:

- `qa` starts recording a macro in the "a register.
- `0` goes to beginning of the line.
- `gU$` uppercases the text from your current location to the end of the line.
- `j` goes down one line.
- `q` stops recording.

To replay it, run `@a`. Just like many other Vim commands, you can pass a count argument to macros. For example, running `3@a` executes the macro three times.

Safety Guard

Macro execution automatically ends when it encounters an error. Suppose you have this text:

```
a. chocolate donut
b. mochi donut
c. powdered sugar donut
d. plain donut
```

If you want to uppercase the first word on each line, this macro should work:

```
qa0W~jq
```

Here's the breakdown of the command above:

- `qa` starts recording a macro in the "a register.
- `0` goes to the beginning of the line.
- `W` goes to the next WORD.
- `~` toggles the case of the character under the cursor.
- `j` goes down one line.
- `q` stops recording.

I prefer to overcount my macro execution than undercount it, so I usually call it ninety-nine times (`99@a`). With this command, Vim does not actually run this macro ninety-nine times. When Vim reaches the last line and runs `j` motion, it finds no more line to go down to, throws an error, and stops the macro execution.

The fact that macro execution stops upon the first error encounter is a good feature, otherwise Vim will continue to execute this macro ninety-nine times even though it already reaches the end of the line.

Command Line Macro

Running `@a` in normal mode is not the only way you can execute macros in Vim. You can also run `:normal @a` command line. `:normal` allows the user to execute any normal mode command passed as argument. In the case above, it is the same as running `@a` from normal mode.

The `:normal` command accepts range as arguments. You can use this to run macro in select ranges. If you want to execute your macro between lines 2 and 3, you can run `:2,3 normal @a`.

Executing A Macro Across Multiple Files

Suppose you have multiple `.txt` files, each contains some texts. Your task is to uppercase the first word only on lines containing the word "donut". Assume you have `0W~j` in register a (the same macro as before). How can you quickly accomplish this?

First file:

```
## savory.txt
a. cheddar jalapeno donut
b. mac n cheese donut
c. fried dumpling
```

Second file:

```
## sweet.txt
a. chocolate donut
b. chocolate pancake
c. powdered sugar donut
```

Third file:

```
## plain.txt
a. wheat bread
b. plain donut
```

Here is how you can do it:

- `:args *.txt` to find all `.txt` files in your current directory.
- `:argdo g/donut/normal @a` executes the global command `g/donut/normal @a` on each file inside `:args`.
- `:argdo update` executes `update` command to save each file inside `:args` when the buffer has been modified.

If you are not familiar with the global command `:g/donut/normal @a`, it executes the command you give (`normal @a`) on lines that match the pattern (`/donut/`). I will go over the global command in a later chapter.

Recursive Macro

You can recursively execute a macro by calling the same macro register while recording that macro. Suppose you have this list again and you need to toggle the case of the first word:

```
a. chocolate donut
b. mochi donut
c. powdered sugar donut
d. plain donut
```

This time, let's do it recursively. Run:

```
qaqqa0W~j@aq
```

Here is the breakdown of the steps:

- `qaq` records an empty macro "a. It is necessary to start with an empty register because when you recursively call the macro, it will run whatever is in that register.
- `qa` starts recording on register a.
- `0` goes to the first character in the current line.
- `W` goes to the next WORD.
- ~ toggles the case of the character under the cursor.
- `j` goes down one line.
- `@a` executes macro "a.
- `q` stops recording.

Now you can just run `@a` and watch Vim execute the macro recursively.

How did the macro know when to stop? When the macro was on the last line, it triedto run `j`, since there was no more line to go to, it stopped the macro execution.

Appending A Macro

If you need to add actions to an existing macro, instead of recreating the macro from scratch, you can append actions to an existing one. In the register chapter, you learned that you can append a named register by using its uppercased symbol. The same rule applies. To append actions to register a macro, use register "A.

Record a macro in register a: `qa0W~q` (this sequence toggles the case of the next WORD in a line). If you want to append a new sequence to also add a dot at the end of the line, run:

```
qAA.<Esc>q
```

The breakdown:

- `qA` starts recording the macro in register "A.
- `A.<Esc>` inserts at the end of the line (here `A` is the insert mode command, not to be confused with the macro "A) a dot, then exits insert mode.
- `q` stops recording macro.

Now when you execute `@a`, it not only toggles the case of the next WORD, it also adds a dot at the end of the line.

Amending A Macro

What if you need to add new actions in the middle of a macro?

Assume that you have a macro that toggles the first actual word and adding a period at the end of the line, `0W~A.<Esc>` in register a. Suppose that between uppercasing the first word and adding a period at the end of the line, you need to add the word "deep fried" right before the word "donut" *(because the only thing better than regular donuts are deep fried donuts).*

I will reuse the text from earlier section:

```
a. chocolate donut
b. mochi donut
c. powdered sugar donut
d. plain donut
```

First, let's call the existing macro (assume you have kept the macro from the previous section in register a) with `:put a`:

```
0W~A.^[
```

What is this `^[`? Didn't you do `0W~A.<Esc>`? Where is the `<Esc>`? `^[` is Vim's *internal code* representation of `<Esc>`. With certain special keys, Vim prints the representation of those keys in the form of internal codes. Some common keys that have internal code representations are `<Esc>`, `<Backspace>`, and `<Enter>`. There are more special keys, but they are not within the scope of this chapter.

Back to the macro, right after the toggle case operator (~), let's add the instructions to go to the end of the line (`$`), go back one word (`b`), go to the insert mode (`i`), type "deep fried " (don't forget the space after "fried "), and exit insert mode (`<Esc>`).

Here is what you will end up with:

```
0W~$bideep fried <Esc>A.^[
```

There is a small problem. Vim does not understand `<Esc>`. You can't literally type `<Esc>`. You will have to write the internal code representation for the `<Esc>` key. While in insert mode, you press `Ctrl-V` followed by `<Esc>`. Vim will print `^[`. `Ctrl-V` is an insert mode operator to insert the next non-digit character *literally*. Your macro code should look like this now:

```
0W~$bideep fried ^[A.^[
```

To add the amended instruction into register a, you can do it the same way as adding a new entry into a named register. At the start of the line, run `"ay$` to store the yanked text in register a.

Now when you execute `@a`, your macro will toggle the case of the first word, add "deep fried " before "donut", and add a "." at the end of the line. Yum!

An alternative way to amend a macro is to use a command line expression. Do `:let @a="`, then do `Ctrl-R Ctrl-R a`, this will literally paste the content of register a. Finally, don't forget to close the double quotes (`"`). You might have something like `:let @a="0W~$bideep fried ^[A.^["`.

Macro Redundancy

You can easily duplicate macros from one register to another. For example, to duplicate a macro in register a to register z, you can do `:let @z = @a`. `@a` represents the content of register a. Now if you run `@z`, it does the exact same actions as `@a`.

I find creating a redundancy useful on my most frequently used macros. In my workflow, I usually record macros in the first seven alphabetical letters (a-g) and I often replace them without much thought. If I move the useful macros towards the end of the alphabets, I can preserve them without worrying that I might accidentally replace them.

Series Vs Parallel Macro

Vim can execute macros in series and parallel. Suppose you have this text:

```
import { FUNC1 } from "library1";
import { FUNC2 } from "library2";
import { FUNC3 } from "library3";
import { FUNC4 } from "library4";
import { FUNC5 } from "library5";
```

If you want to record a macro to lowercase all the uppercased "FUNC", this macro should work:

```
qa0f{gui{jq
```

The breakdown:

- `qa` starts recording in register a.
- `0` goes to first line.
- `f{` finds the first instance of "{".
- `gui{` lowercases (`gu`) the text inside the bracket text-object (`i{`).
- `j` goes down one line.
- `q` stops macro recording.

Now you can run `99@a` to execute it on the remaining lines. However, what if you have this import expression inside your file?

```
import { FUNC1 } from "library1";
import { FUNC2 } from "library2";
import { FUNC3 } from "library3";
import foo from "bar";
import { FUNC4 } from "library4";
import { FUNC5 } from "library5";
```

Running `99@a`, only executes the macro three times. It does not execute the macro on last two lines because the execution fails to run `f{` on the "foo" line. This is expected when running the macro in series. You can always go to the next line where "FUNC4" is and replay that macro again. But what if you want to get everything done in one go?

Run the macro in parallel.

Recall from earlier section that macros can be executed using the command line command `:normal` (ex: `:3,5 normal @a` executes macro "a on lines 3-5). If you run `:1,$ normal @a`, you will see that the macro is being executed on all lines except the "foo" line. It works!

Although internally Vim does not actually run the macros in parallel, outwardly, it behaves like it. Vim executes `@a` *independently* on each line from the first to the last line (`1,$`). Since Vim executes these macros independently, each line does not know that one of the macro executions had failed on the "foo" line.

Learn Macros The Smart Way

Many things you do in editing are repetitive. To get better at editing, get into the habit of detecting repetitive actions. Use macros (or dot command) so you don't have to perform the same action twice. Almost everything that you can do in Vim can be replicated with macros.

In the beginning, I find it very awkward to write macros, but don't give up. With enough practice, you will get into the habit of automating everything.

You might find it helpful to use mnemonics to help remember your macros. If you have a macro that creates a function, use the "f register (`qf`). If you have a macro for numerical operations, the "n register should work (`qn`). Name it with the *first named register* that comes to your mind when you think of that operation. I also find that the "q register makes a good default macro register because `qq` requires less brain power to come up with. Lastly, I also like to increment my macros in alphabetical orders, like `qa`, then `qb`, then `qc`, and so on.

Find a method that works best for you.

Ch10. Undo

We all make all sorts of typing mistakes. That's why undo is an essential feature in any modern software. Vim's undo system is not only capable of undoing and redoing simple mistakes, but also accessing different text states, giving you control to all the texts you have ever typed. In this chapter, you will learn how to undo, redo, navigate an undo branch, persist undo, and travel across time.

Undo, Redo, And UNDO

To perform a basic undo, you can use `u` or run `:undo`.

If you have this text (note the empty line below "one"):

```
one

```

Then you add another text:

```
one
two
```

If you press `u`, Vim undoes the text "two".

How does Vim know how much to undo? Vim undoes a single "change" at a time, similar to a dot command's change (unlike the dot command, command-line command also count as a change).

To redo the last change, press `Ctrl-R` or run `:redo`. After you undo the text above to remove "two", running `Ctrl-R` will get the removed text back.

Vim also has UNDO that you can run with `U`. It undoes all latest changes.

How is `U` different from `u`? First, `U` removes *all* the changes on the latest changed line, while `u` only removes one change at a time. Second, while doing `u` does not count as a change, doing `U` counts as a change.

Back to this example:

```
one
two
```

Change the second line to "three":

```
one
three
```

Change the second line again and replace it with "four":

```
one
four
```

If you press `u`, you will see "three". If you press `u` again, you'll see "two". If instead of pressing `u` when you still had the text "four", you had pressed `U`, you will see:

```
one
```

`U` bypasses all the intermediary changes and goes to the original state when you started (an empty line). In addition, since UNDO actually creates a new change in Vim, you can UNDO your UNDO. `U` followed by `U` will undo itself. You can press `U`, then `U`, then `U`, etc. You will see the same two text states toggling back and forth.

I personally do not use `U` because it is hard to remember the original state (I seldom ever need it).

Vim sets a maximum number of how many times you can undo in `undolevels` option variable. You can check it with `:echo &undolevels`. I have mine set to be 1000. To change yours to 1000, run `:set undolevels=1000`. Feel free to set it to any number you like.

Breaking The Blocks

I mentioned earlier that `u` undoes a single "change" similar to the dot command's change: the texts inserted from when you enter the insert mode until you exit it count as a change.

If you do `ione two three<Esc>` then press `u`, Vim removes the entire "one two three" text because the whole thing counts as a change. This is not a big deal if you have written short texts, but what if you have written several paragraphs within one insert mode session without exiting and later you realized you made a mistake? If you press `u`, everything you had written would be removed. Wouldn't it be useful if you can press `u` to remove only a section of your text?

Luckily, you can break the undo blocks. When you are typing in insert mode, pressing `Ctrl-G u` creates an undo breakpoint. For example, if you do `ione <Ctrl-G u>two <Ctrl-G u>three<Esc>`, then press `u`, you will only lose the text "three" (press `u` one more time to remove "two"). When you write a long text, use `Ctrl-G u` strategically. The end of each sentence, between two paragraphs, or after each line of code are prime locations to add undo breakpoints to make it easier to undo your mistakes if you ever make one.

It is also useful to create an undo breakpoint when deleting chunks in insert mode with `Ctrl-W` (delete the word before the cursor) and `Ctrl-U` (delete all text before the cursor). A friend suggested to use the following maps:

```
inoremap <c-u> <c-g>u<c-u>
inoremap <c-w> <c-g>u<c-w>
```

With these, you can easily recover the deleted texts.

Undo Tree

Vim stores every change ever written in an undo tree. Start a new empty file. Then add a new text:

```
one
```

Add a new text:

```
one
two
```

Undo once:

```
one
```

Add a different text:

```
one
three
```

Undo again:

```
one
```

And add another different text:

```
one
four
```

Now if you undo, you will lose the text “four” you just added:

```
one
```

If you undo one more time:

You will lose the text "one". In most text editor, getting the texts "two" and "three" back would have been impossible, but not with Vim! Press `g+` and you'll get your text "one" back:

```
one
```

Type `g+` again and you will see an old friend:

```
one
two
```

Let's keep going. Press `g+` again:

```
one
three
```

Press `g+` one more time:

```
one
four
```

In Vim, every time you press `u` and then make a different change, Vim stores the previous state's text by creating an "undo branch". In this example, after you typed "two", then pressed `u`, then typed "three", you created an leaf branch that stores the state containing the text "two". At that moment, the undo tree contained at least two leaf nodes: the main node containing the text "three" (most recent) and the undo branch node containing the text "two". If you had done another undo and typed the text "four", you would have at three nodes: a main node containing the text "four" and two nodes containing the texts "three" and "two".

To traverse each undo tree nodes, you can use `g+` to go to a newer state and `g-` to go to an older state. The difference between `u`, `Ctrl-R`, `g+`, and `g-` is that both `u` and `Ctrl-R` traverse only the *main* nodes in undo tree while `g+` and `g-` traverse *all* nodes in the undo tree.

Undo tree is not easy to visualize. I find vim-mundo[25] plugin to be very useful to help visualize Vim's undo tree. Give it some time to play around with it.

Persistent Undo

If you start Vim, open a file, and immediately press `u`, Vim will probably display "*Already at oldest change*" warning. There is nothing to undo because you haven't made any changes.

To rollover the undo history from the last editing session, Vim can preserve your undo history with an undo file with `:wundo`.

Create a file `mynumbers.txt`. Type:

[25] https://github.com/simnalamburt/vim-mundo

```
one
```

Then type another line (make sure each line counts as a change):

```
one
two
```

Type another line:

```
one
two
three
```

Now create your undo file with `:wundo {my-undo-file}`. If you need to overwrite an existing undo file, you can add `!` after `wundo`.

```
:wundo! mynumbers.undo
```

Then exit Vim.

By now you should have `mynumbers.txt` and `mynumbers.undo` files in your directory. Open up `mynumbers.txt` again and try pressing `u`. You can't. You haven't made any changes since you opened the file. Now load your undo history by reading the undo file with `:rundo`:

```
:rundo mynumbers.undo
```

Now if you press `u`, Vim removes "three". Press `u` again to remove "two". It is like you never even closed Vim!

If you want to have an automatic undo persistence, one way to do it is by adding these in vimrc:

```
set undodir=~/.vim/undo_dir
set undofile
```

The setting above will put all the undofile in one centralized directory, the `~/.vim` directory. The name `undo_dir` is arbitrary. `set undofile` tells Vim to turn on `undofile` feature because it is off by default. Now whenever you save, Vim automatically creates and updates the relevant file inside the `undo_dir` directory (make sure that you create the actual `undo_dir` directory inside `~/.vim` directory before running this).

Time Travel

Who says that time travel doesn't exist? Vim can travel to a text state in the past with `:earlier` command-line command.

If you have this text:

```
one
```

Then later you add:

```
one
two
```

If you had typed “two” less than ten seconds ago, you can go back to the state where “two” didn’t exist ten seconds ago with:

```
:earlier 10s
```

You can use `:undolist` to see when the last change was made. `:earlier` also accepts different arguments:

```
:earlier 10s    Go to the state 10 seconds before
:earlier 10m    Go to the state 10 minutes before
:earlier 10h    Go to the state 10 hours before
:earlier 10d    Go to the state 10 days before
```

In addition, it also accepts a regular `count` as argument to tell Vim to go to the older state `count` times. For example, if you do `:earlier 2`, Vim will go back to an older text state two changes ago. It is the same as doing `g-` twice. You can also tell it to go to the older text state `10` saves ago with `:earlier 10f`.

The same set of arguments work with `:earlier` counterpart: `:later`.

```
:later 10s    go to the state 10 seconds later
:later 10m    go to the state 10 minutes later
:later 10h    go to the state 10 hours later
:later 10d    go to the state 10 days later
:later 10     go to the newer state 10 times
:later 10f    go to the state 10 saves later
```

Learn Undo The Smart Way

`u` and `Ctrl-R` are two indispensable Vim commands for correcting mistakes. Learn them first. Next, learn how to use `:earlier` and `:later` using the time arguments first. After that, take your time to understand the undo tree. The vim-mundo[26] plugin helped me a lot. Type along the texts in this chapter and check the undo tree as you make each change. Once you grasp it, you will never see undo system the same way again.

Prior to this chapter, you learned how to find any text in a project space, with undo, you can now find any text in a time dimension. You are now able to search for any text by its location and time written. You have achieved Vim-omnipresence.

[26] https://github.com/simnalamburt/vim-mundo

Ch11. Visual Mode

Highlighting and applying changes to a body of text is a common feature in many text editors and word processors. Vim can do this using visual mode. In this chapter, you will learn how to use the visual mode to manipulate texts efficiently.

The Three Types Of Visual Modes

Vim has three different visual modes. They are:

```
v         Character-wise visual mode
V         Line-wise visual mode
Ctrl-v    Block-wise visual mode
```

If you have the text:

```
one
two
three
```

Character-wise visual mode works with individual characters. Press `v` on the first character. Then go down to the next line with `j`. It highlights all texts from "one" up to your cursor location. If you press `gU`, Vim uppercases the highlighted characters.

Line-wise visual mode works with lines. Press `V` and watch Vim selects the entire line your cursor is on. Just like character-wise visual mode, if you run `gU`, Vim uppercases the highlighted characters.

Block-wise visual mode works with rows and columns. It gives you more freedom of movement than the other two modes. If you press `Ctrl-v`, Vim highlights the character under the cursor just like character-wise visual mode, except instead of highlighting each character until the end of the line before going down to the next line, it goes to the next line with minimal highlighting. Try moving around with `h/j/k/l` and watch the cursor moves.

On the bottom left of your Vim window, you will see either `-- VISUAL --`, `-- VISUAL LINE --`, or `-- VISUAL BLOCK --` displayed to indicate which visual mode you are in.

While you are inside a visual mode, you can switch to another visual mode by pressing either `v`, `V`, or `Ctrl-v`. For example, if you are in line-wise visual mode and you want to switch to block-wise visual mode, run `Ctrl-v`. Try it!

There are three ways to exit the visual mode: `<Esc>`, `Ctrl-C`, and the same key as your current visual mode. What the latter means is if you are currently in the line-wise visual mode (`V`), you can exit it by pressing `V` again. If you are in the character-wise visual mode, you can exit it by pressing `v`.

There is actually one more way to enter the visual mode:

```
gv    Go to the previous visual mode
```

It will start the same visual mode on the same highlighted text block as you did last time.

Visual Mode Navigation

While in a visual mode, you can expand the highlighted text block with Vim motions.

Let's use the same text you used earlier:

```
one
two
three
```

This time let's start from the line "two". Press `v` to go to the character-wise visual mode (here the square brackets `[]` represents the character highlights):

```
one
[t]wo
three
```

Press `j` and Vim will highlight all the text from the line "two" down to the first character of the line "three".

```
one
[two
t]hree
```

Assume from this position, you want to add the line "one" too. If you press `k`, to your dismay, the highlight moves away from the line "three".

```
one
[t]wo
three
```

Is there a way to freely expand visual selection to go to any direction you want? Definitely. Let's back up a little bit to where you have the line "two" and "three" highlighted.

```
one
[two
t]hree    <-- cursor
```

Visual highlight follows the cursor movement. If you want to expand it upward to line “one”, you need to move the cursor up to the line “two”. Right now the cursor is on the line “three”. You can toggle the cursor location with either `o` or `0`.

```
one
[two     <-- cursor
t]hree
```

Now when you press `k`, it no longer reduces the selection, but expands it upward.

```
[one
two
t]hree
```

With `o` or `0` in visual mode, the cursor jumps from the beginning to the end of the highlighted block, allowing you to expand the highlight area.

Visual Mode Grammar

The visual mode shares many operations with normal mode.

For example, if you have the following text and you want to delete the first two lines from visual mode:

```
one
two
three
```

Highlight the lines “one” and “two” with the line-wise visual mode (`V`):

```
[one
two]
three
```

Pressing `d` will delete the selection, similar to normal mode. Notice the grammar rule from normal mode, verb + noun, does not apply. The same verb is still there (`d`), but there is no noun in visual mode. The grammar rule in visual mode is noun + verb, where noun is the highlighted text. Select the text block first, then the command follows.

In normal mode, there are some commands that do not require a motion, like `x` to delete a single character under the cursor and `r` to replace the character under the cursor (`rx` replaces the character under the cursor with "x"). In visual mode, these commands are now being applied to the entire highlighted text instead of a single character. Back at the highlighted text:

```
[one
two]
three
```

Running `x` deletes all highlighted texts.

You can use this behavior to quickly create a header in markdown text. Suppose you need to quickly turn the following text into a first-level markdown header ("`===`"):

```
Chapter One
```

First, copy the text with `yy`, then paste it with `p`:

```
Chapter One
Chapter One
```

Now go to the second line, select it with line-wise visual mode:

```
Chapter One
[Chapter One]
```

A first-level header is a series of "`=`" below a text. Run `r=`, voila! This saves you from typing "`=`" manually.

```
Chapter One
===========
```

To learn more about operators in visual mode, check out `:h visual-operators`.

Visual Mode And Command-line Commands

You can selectively apply command-line commands on a highlighted text block. If you have these statements and you want to substitute "const" with "let" only on the first two lines:

```
const one = "one";
const two = "two";
const three = "three";
```

Highlight the first two lines with *any* visual mode and run the substitute command `:s/const/let/g`:

```
let one = "one";
let two = "two";
const three = "three";
```

Notice I said you can do this with *any* visual mode. You do not have to highlight the entire line to run the command on that line. As long as you select at least a character on each line, the command is applied.

Adding Text On Multiple Lines

You can add text on multiple lines in Vim using the block-wise visual mode. If you need to add a semicolon at the end of each line:

```
const one = "one"
const two = "two"
const three = "three"
```

With your cursor on the first line:

- Run block-wise visual mode and go down two lines (`Ctrl-v jj`).
- Highlight to the end of the line (`$`).
- Append (`A`) then type ";".
- Exit visual mode (`<Esc>`).

You should see the appended ";" on each line now. Pretty cool! There are two ways to enter the insert mode from block-wise visual mode: `A` to enter the text after the cursor or `I` to enter the text before the cursor. Do not confuse them with `A` (append text at the end of the line) and `I` (insert text before the first non-blank line) from normal mode.

Alternatively, you can also use the `:normal` command to add text on multiple lines:

- Highlight all 3 lines (`vjj`).
- Type `:normal! A;`.

Remember, `:normal` command executes normal mode commands. You can instruct it to run `A;` to append text ";" at the end of the line.

Incrementing Numbers

Vim has `Ctrl-X` and `Ctrl-A` commands to decrement and increment numbers. When used with visual mode, you can increment numbers across multiple lines.

If you have these HTML elements:

```
<div id="app-1"></div>
<div id="app-1"></div>
<div id="app-1"></div>
<div id="app-1"></div>
<div id="app-1"></div>
```

It is a bad practice to have several ids having the same name, so let's increment them to make them unique:

- Move your cursor to the "1" on the second line.
- Start block-wise visual mode and go down 3 lines (`Ctrl-v 3j`). This highlights the remaining "1"s. Now all "1" should be highlighted.
- Run `g Ctrl-a`.

You should see this result:

```
<div id="app-1"></div>
<div id="app-2"></div>
<div id="app-3"></div>
<div id="app-4"></div>
<div id="app-5"></div>
```

`g Ctrl-a` increments numbers on multiple lines. `Ctrl-X/Ctrl-A` can increment letters too, with the number formats option:

```
set nrformats+=alpha
```

The `nrformats` option instructs Vim which bases are considered as "numbers" for `Ctrl-A` and `Ctrl-X` to increment and decrement. By adding `alpha`, an alphabetical character is now considered as a number. If you have the following HTML elements:

```
<div id="app-a"></div>
<div id="app-a"></div>
<div id="app-a"></div>
<div id="app-a"></div>
<div id="app-a"></div>
```

Put your cursor on the second "app-a". Use the same technique as above (`Ctrl-v 3j` then `g Ctrl-a`) to increment the ids.

```
<div id="app-a"></div>
<div id="app-b"></div>
<div id="app-c"></div>
<div id="app-d"></div>
<div id="app-e"></div>
```

Selecting The Last Visual Mode Area

Earlier in this chapter I mentioned that `gv` can quickly highlight the last visual mode highlight. You can also go to the location of the start and the end of the last visual mode with these two special marks:

```
`<    Go to the last place of the previous visual mode highlight
`>    Go to the first place of the previous visual mode highlight
```

Earlier, I also mentioned that you can selectively execute command-line commands on a highlighted text, like `:s/const/let/g`. When you did that, you'd see this below:

```
:`<,`>s/const/let/g
```

You were actually executing a *ranged* `s/const/let/g` command (with the two marks as the addresses). Interesting!

You can always edit these marks anytime you wish. If instead you needed to substitute from the start of the highlighted text to the end of the file, you just change the command to:

```
:`<,$s/const/let/g
```

Entering Visual Mode From Insert Mode

You can also enter visual mode from the insert mode. To go to character-wise visual mode while you are in insert mode:

```
Ctrl-O v
```

Recall that running `Ctrl-O` while in the insert mode lets you to execute a normal mode command. While in this normal-mode-command-pending mode, run `v` to enter character-wise visual mode. Notice that on the bottom left of the screen, it says `--(insert) VISUAL--`. This trick works with any visual mode operator: `v`, `V`, and `Ctrl-v`.

Select Mode

Vim has a mode similar to visual mode called the select mode. Like the visual mode, it also has three different modes:

```
gh         Character-wise select mode
gH         Line-wise select mode
gCtrl-h    Block-wise select mode
```

Select mode emulates a regular editor's text highlighting behavior closer than Vim's visual mode does.

In a regular editor, after you highlight a text block and type a letter, say the letter "y", it will delete the highlighted text and insert the letter "y". If you highlight a line with line-wise select mode (`gH`) and type "y", it will delete the highlighted text and insert the letter "y".

Contrast this select mode with visual mode: if you highlight a line of text with line-wise visual mode (`V`) and type "y", the highlighted text will not be deleted and replaced by the literal letter "y", it will be yanked. You can't execute normal mode commands on highlighted text in select mode.

I personally never used select mode, but it's good to know that it exists.

Learn Visual Mode The Smart Way

The visual mode is Vim's representation of the text highlighting procedure.

If you find yourself using visual mode operation far more often than normal mode operations, be careful. This is an anti-pattern. It takes more keystrokes to run a visual mode operation than its normal mode counterpart. For example, if you need to delete an inner word, why use four keystrokes, `viwd` (visually highlight an inner word then delete), if you can accomplish it with just three keystrokes (`diw`)? The latter is more direct and concise. Of course, there will be times when visual modes are appropriate, but in general, favor a more direct approach.

Ch12. Search And Substitute

This chapter covers two separate but related concepts: search and substitute. Often when editing, you need to search multiple texts based on their least common denominator patterns. By learning how to use regular expressions in search and substitute instead of literal strings, you will be able to target any text quickly.

As a side note, in this chapter, I will use `/` when talking about search. Everything you can do with `/` can also be done with `?`.

Smart Case Sensitivity

It can be tricky trying to match the case of the search term. If you are searching for the text "Learn Vim", you can easily mistype the case of one letter and get a false search result. Wouldn't it be easier and safer if you can match any case? This is where the option `ignorecase` shines. Just add `set ignorecase` in your vimrc and all your search terms become case insensitive. Now you don't have to do `/Learn Vim` anymore, `/learn vim` will work.

However, there are times when you need to search for a case specific phrase. One way to do that is to turn off `ignorecase` option by running `set noignorecase`, but that is a lot of work to turn on and off each time you need to search for a case sensitive phrase.

To avoid toggling `ignorecase`, Vim has a `smartcase` option to search for case insensitive string if the search pattern *contains at least one uppercase character*. You can combine both `ignorecase` and `smartcase` to perform a case insensitive search when you enter all lowercase characters and a case sensitive search when you enter one or more uppercase characters.

Inside your vimrc, add:

```
set ignorecase smartcase
```

If you have these texts:

```
hello
HELLO
Hello
```

- `/hello` matches "hello", "HELLO", and "Hello".
- `/HELLO` matches only "HELLO".
- `/Hello` matches only "Hello".

There is one downside. What if you need to search for only a lowercase string? When you do `/hello`, Vim now does case insensitive search. You can use `\C` pattern anywhere in your search term to tell Vim that the subsequent search term will be case sensitive. If you do `/\Chello`, it will strictly match "hello", not "HELLO" or "Hello".

First And Last Character In A Line

You can use `^` to match the first character in a line and `$` to match the last character in a line.

If you have this text:

```
hello hello
```

You can target the first "hello" with `/^hello`. The character that follows `^` must be the first character in a line. To target the last "hello", run `/hello$`. The character before `$` must be the last character in a line.

If you have this text:

```
hello hello friend
```

Running `/hello$` will not match anything because "friend" is the last term in that line, not "hello".

Repeating Search

You can repeat the previous search with `//`. If you have just searched for `/hello`, running `//` is equivalent to running `/hello`. This shortcut can save you some keystrokes especially if you just searched for a long string. Also recall that you can use `n` and `N` to repeat the last search with the same direction and opposite direction, respectively.

What if you want to quickly recall *n* last search term? You can quickly traverse the search history by first pressing `/`, then press `up/down` arrow keys (or `Ctrl-N`/`Ctrl-P`) until you find the search term you need. To see all your search history, you can run `:history /`.

When you reach the end of a file while searching, Vim throws an error: `"Search hit the BOTTOM without match for: {your-search}"`. Sometimes this can be a good safeguard from oversearching, but other times you want to cycle the search back to the top again. You can use the `set wrapscan` option to make Vim to search back at the top of the file when you reach the end of the file. To turn this feature off, do `set nowrapscan`.

Searching For Alternative Words

It is common to search for multiple words at once. If you need to search for *either* "hello vim" or "hola vim", but not "salve vim" or "bonjour vim", you can use the `|` pattern.

Given this text:

```
hello vim
hola vim
salve vim
bonjour vim
```

To match both "hello" and "hola", you can do `/hello\|hola`. You have to escape (`\`) the or (`|`) operator, otherwise Vim will literally search for the string "|".

If you don't want to type `\|` every time, you can use the `magic` syntax (`\v`) at the start of the search: `/\vhello|hola`. I will not cover `magic` in this guide, but with `\v`, you don't have to escape special characters anymore. To learn more about `\v`, feel free to check out `:h \v`.

Setting The Start And End Of A Match

Maybe you need to search for a text that is a part of a compound word. If you have these texts:

```
11vim22
vim22
11vim
vim
```

If you need to select "vim" but only when it starts with "11" and ends with "22", you can use `\zs` (starting match) and `\ze` (ending match) operators. Run:

```
/11\zsvim\ze22
```

Vim still has to match the entire pattern "11vim22", but only highlights the pattern sandwiched between `\zs` and `\ze`. Another example:

```
foobar
foobaz
```

If you need to match the "foo" in "foobaz" but not in "foobar", run:

```
/foo\zebaz
```

Searching Character Ranges

All your search terms up to this point have been a literal word search. In real life, you may have to use a general pattern to find your text. The most basic pattern is the character range, `[ ]`.

If you need to search for any digit, you probably don't want to type `/0\|1\|2\|3\|4\|5\|6\|7\|8\|9\|0` every single time. Instead, use `/[0-9]` to match for a single digit. The `0-9` expression represents a range of numbers 0-9 that Vim will try to match, so if you are looking for digits between 1 to 5 instead, use `/[1-5]`.

Digits are not the only data types Vim can look up. You can also do `/[a-z]` to search for lowercase alphas and `/[A-Z]` to search for uppercase alphas.

You can combine these ranges together. If you need to search for digits 0-9 and both lowercase and uppercase alphas from "a" to "f" (like a hex), you can do `/[0-9a-fA-F]`.

To do a negative search, you can add `^` inside the character range brackets. To search for a non-digit, run `/[^0-9]`. Vim will match any character as long as it is not a digit. Beware that the caret (`^`) inside the range brackets is different from the beginning-of-a-line caret (ex: `/^hello`). If a caret is outside of a pair of brackets and is the first character in the search term, it means "the first character in a line". If a caret is inside a pair of brackets and it is the first character inside the brackets, it means a negative search operator. `/^abc` matches the first "abc" in a line and `/[^abc]` matches any character except for an "a", "b", or "c".

Searching For Repeating Characters

If you need to search for double digits in this text:

```
1aa
11a
111
```

You can use `/[0-9][0-9]` to match a two-digit character, but this method is unscalable. What if you need to match twenty digits? Typing `[0-9]` twenty times is not a fun experience. That's why you need a `count` argument.

You can pass `count` to your search. It has the following syntax:

```
{n,m}
```

By the way, these `count` braces need to be escaped when you use them in Vim. The `count` operator is placed after a single character you want to increment.

Here are the four different variations of the `count` syntax:

- `{n}` is an exact match. `/[0-9]\{2\}` matches the two digit numbers: "11" and the "11" in "111".
- `{n,m}` is a range match. `/[0-9]\{2,3\}` matches between 2 and 3 digit numbers: "11" and "111".
- `{,m}` is an up-to match. `/[0-9]\{,3\}` matches up to 3 digit numbers: "1", "11", and "111".
- `{n,}` is an at-least match. `/[0-9]\{2,\}` matches at least a 2 or more digit numbers: "11" and "111".

The count arguments `\{0,\}` (zero or more) and `\{1,\}` (one or more) are common search patterns and Vim has special operators for them: `*` and `+` (`+` needs to be escaped while `*` works fine without the escape). If you do `/[0-9]*`, it is the same as `/[0-9]\{0,\}`. It searches for zero or more digits. It will match "", "1", "123". By the way, it will also match non-digits like "a", because there is technically zero digit in the letter "a". Think carefully before using `*`. If you do `/[0-9]\+`, it is the same as `/[0-9]\{1,\}`. It searches for one or more digits. It will match "1" and "12".

Predefined Character Ranges

Vim has predefined ranges for common characters like digits and alphas. I will not go through every single one here, but you can find the full list inside `:h /character-classes`. Here are the useful ones:

```
\d    Digit [0-9]
\D    Non-digit [^0-9]
\s    Whitespace character (space and tab)
\S    Non-whitespace character (everything except space and tab)
\w    Word character [0-9A-Za-z_]
\l    Lowercase alphas [a-z]
\u    Uppercase character [A-Z]
```

You can use them like you would use character ranges. To search for any single digit, instead of using `/[0-9]`, you can use `/\d` for a more concise syntax.

Search Example: Capturing A Text Between A Pair Of Similar Characters

If you want to search for a phrase surrounded by a pair of double quotes:

```
"Vim is awesome!"
```

Run this:

```
/"[^"]\+"
```

Let's break it down:

- `"` is a literal double quote. It matches the first double quote.
- `[^"]` means any character except for a double quote. It matches any alphanumeric and whitespace character as long as it is not a double quote.
- `\+` means one or more. Since it is preceded by `[^"]`, Vim looks for one or more character that is not a double quote.
- `"` is a literal double quote. It matches the closing double quote.

When Vim sees the first `"`, it begins the pattern capture. The moment it sees the second double quote in a line, it matches the second `"` pattern and stops the pattern capture. Meanwhile, all non-double-quote characters inbetween are captured by the `[^"]\+` pattern, in this case, the phrase `Vim is awesome!`. This is a common pattern to capture a phrase surrounded by a pair of similar delimiters.

- To capture a phrase surrounded by single quotes, you can use `/'[^']\+'`.
- To capture a phrase surrounded by zeroes, you can use `/0[^0]\+0`.

Search Example: Capturing A Phone Number

If you want to match a US phone number separated by a hyphen (-), like `123-456-7890`, you can use:

```
/\d\{3\}-\d\{3\}-\d\{4\}
```

US Phone number consists of a set of three digit number, followed by another three digits, and finally by four digits. Let's break it down:

- `\d\{3\}` matches a digit repeated exactly three times
- `-` is a literal hyphen

You can avoid typing escapes with `\v`:

```
/\v\d{3}-\d{3}-\d{4}
```

This pattern is also useful to capture any repeating digits, such as IP addresses and zip codes.

That covers the search part of this chapter. Now let's move to substitution.

Basic Substitution

Vim's substitute command is a useful command to quickly find and replace any pattern. The substitution syntax is:

```
:s/{old-pattern}/{new-pattern}/
```

Let's start with a basic usage. If you have this text:

```
vim is good
```

Let's substitute "good" with "awesome" because Vim is awesome. Run `:s/good/awesome/`. You should see:

```
vim is awesome
```

Repeating The Last Substitution

You can repeat the last substitute command with either the normal command `&` or by running `:s`. If you have just run `:s/good/awesome/`, running either `&` or `:s` will repeat it.

Also, earlier in this chapter I mentioned that you can use `//` to repeat the previous search pattern. This trick works with the substitution command. If `/good` was done recently and you leave the first substitute pattern argument blank, like in `:s//awesome/`, it works the same as running `:s/good/awesome/`.

Substitution Range

Just like many Ex commands, you can pass a range argument into the substitute command. The syntax is:

```
:[range]s/old/new/
```

If you have these expressions:

```
let one = 1;
let two = 2;
let three = 3;
let four = 4;
let five = 5;
```

To substitute the "let" into "const" on lines three to five, you can do:

```
:3,5s/let/const/
```

Here are some range variations you can pass:

- `:,3/let/const/` - if nothing is given before the comma, it represents the current line. Substitute from current line to line 3.
- `:1,s/let/const/` - if nothing is given after the comma, it also represents the current line. Substitute from line 1 to current line.
- `:3s/let/const/` - if only one value is given as range (no comma), it does substitution on that line only.

In Vim, `%` usually means the entire file. If you run `:%s/let/const/`, it will do substitution on all lines. Keep in mind of this range syntax. Many command-line commands that you will learn in the upcoming chapters will follow this form.

Pattern Matching

The next few sections will cover basic regular expressions. A strong pattern knowledge is essential to master the substitute command.

If you have the following expressions:

```
let one = 1;
let two = 2;
let three = 3;
let four = 4;
let five = 5;
```

To add a pair of double quotes around the digits:

```
:%s/\d/"\0"/
```

The result:

```
let one = "1";
let two = "2";
let three = "3";
let four = "4";
let five = "5";
```

Let's break down the command:

- `:%s` targets the entire file to perform substitution.
- `\d` is Vim's predefined range for digits (similar to using `[0-9]`).
- `"\0"` here the double quotes are literal double quotes. `\0` is a special character representing "the whole matched pattern". The matched pattern here is a single digit number, `\d`.

Alternatively, `&` also represents the whole matched pattern like `\0`. `:s/\d/"&"/` would have also worked.

Let's consider another example. Given these expressions and you need to swap all the "let" with the variable names.

```
one let = "1";
two let = "2";
three let = "3";
four let = "4";
five let = "5";
```

To do that, run:

```
:%s/\(\w\+\) \(\w\+\)/\2 \1/
```

The command above contains too many backslashes and is hard to read. In this case it is more convenient to use the `\v` operator:

```
:%s/\v(\w+) (\w+)/\2 \1/
```

The result:

```
let one = "1";
let two = "2";
let three = "3";
let four = "4";
let five = "5";
```

Great! Let's break down that command:

- `:%s` targets all the lines in the file to perform substitution.
- `(\w+) (\w+)` is a group match. `\w` is one of Vim's predefined ranges for a word character (`[0-9A-Za-z_]`). The `( )` surrounding it captures a word character match in a group. Notice the space between the two groupings. `(\w+) (\w+)` captures two groups. The first group captures "one" and the second group captures "two".

- `\2 \1` returns the captured group in a reversed order. `\2` contains the captured string "let" and `\1` the string "one". Having `\2 \1` returns the string "let one".

Recall that `\0` represents the entire matched pattern. You can break the matched string into smaller groups with `( )`. Each group is represented by `\1`, `\2`, `\3`, etc.

Let's do one more example to solidify this group match concept. If you have these numbers:

```
123
456
789
```

To reverse the order, run:

```
:%s/\v(\d)(\d)(\d)/\3\2\1/
```

The result is:

```
321
654
987
```

Each `(\d)` matches each digit and creates a group. On the first line, the first `(\d)` has a value of 1, the second `(\d)` has a value of 2, and the third `(\d)` has a value of 3. They are stored in the variables `\1`, `\2`, and `\3`. In the second half of your substitution, the new pattern `\3\2\1` results in the "321" value on line one.

If you had run this instead:

```
:%s/\v(\d\d)(\d)/\2\1/
```

You would have gotten a different result:

```
312
645
978
```

This is because you now only have two groups. The first group, captured by `(\d\d)`, is stored within `\1` and has the value of 12. The second group, captured by `(\d)`, is stored inside `\2` and has the value of 3. `\2\1` then, returns 312.

Substitution Flags

If you have the sentence:

```
chocolate pancake, strawberry pancake, blueberry pancake
```

To substitute all the pancakes into donuts, you cannot just run:

```
:s/pancake/donut
```

The command above will only substitute the first match, giving you:

```
chocolate donut, strawberry pancake, blueberry pancake
```

There are two ways to solve this. You can either run the substitute command twice more or you can pass it a global (`g`) flag to substitute all of the matches in a line.

Let's talk about the global flag. Run:

```
:s/pancake/donut/g
```

Vim substitutes all pancakes with donuts in one swift command. The global command is one of the several flags the substitute command accepts. You pass flags at the end of the substitute command. Here is a list of useful flags:

```
&    Reuse the flags from the previous substitute command.
g    Replace all matches in the line.
c    Ask for substitution confirmation.
e    Prevent error message from displaying when substitution fails.
i    Perform case insensitive substitution.
I    Perform case sensitive substitution.
```

There are more flags that I do not list above. To read about all the flags, check out `:h s_flags`.

By the way, the repeat-substitution commands (`&` and `:s`) do not retain the flags. Running `&` will only repeat `:s/pancake/donut/` without `g`. To quickly repeat the last substitute command with all the flags, run `:&&`.

Changing The Delimiter

If you need to replace a URL with a long path:

```
https://mysite.com/a/b/c/d/e
```

To substitute it with the word "hello", run:

```
:s/https:\/\/mysite.com\/a\/b\/c\/d\/e/hello/
```

However, it is hard to tell which forward slashes (`/`) are part of the substitution pattern and which ones are the delimiters. You can change the delimiter with any single-byte characters (except for alphabets, numbers, or `"`, `|`, and `\`). Let's replace them with `+`. The substitution command above then can be rewritten as:

```
:s+https:\/\/mysite.com\/a\/b\/c\/d\/e+hello+
```

It is now easier to see where the delimiters are.

Special Replace

You can also modify the case of the text you are substituting. Given the following expressions and your task is to uppercase the variables "one", "two", "three", etc.

```
let one = "1";
let two = "2";
let three = "3";
let four = "4";
let five = "5";
```

Run:

```
:%s/\v(\w+) (\w+)/\1 \U\2/
```

You will get:

```
let ONE = "1";
let TWO = "2";
let THREE = "3";
let FOUR = "4";
let FIVE = "5";
```

The breakdown:

- `(\w+) (\w+)` captures the first two matched groups, such as "let" and "one".
- `\1` returns the value of the first group, "let".
- `\U\2` uppercases (`\U`) the second group (`\2`).

The trick of this command is the expression `\U\2`. `\U` instructs the following character to be uppercased.

Let's do one more example. Suppose you are writing a Vim guide and you need to capitalize the first letter of each word in a line.

```
vim is the greatest text editor in the whole galaxy
```

You can run:

```
:s/\<./\U&/g
```

The result:

```
Vim Is The Greatest Text Editor In The Whole Galaxy
```

Here is the breakdowns:

- `:s` substitutes the current line.
- `\<.` is comprised of two parts: `\<` to match the start of a word and `.` to match any character. `\<` operator makes the following character to be the first character of a word. Since `.` is the next character, it will match the first character of any word.
- `\U&` uppercases the subsequent symbol, `&`. Recall that `&` (or `\0`) represents the whole match. It matches the first character of any word.
- `g` the global flag. Without it, this command only substitutes the first match. You need to substitute every match on this line.

To learn more of substitution's special replace symbols like `\U`, check out `:h sub-replace-special`.

Alternative Patterns

Sometimes you need to match multiple patterns simultaneously. If you have the following greetings:

```
hello vim
hola vim
salve vim
bonjour vim
```

You need to substitute the word "vim" with "friend" but only on the lines containing the word "hello" or "hola". Recall from earlier this chapter, you can use `|` for multiple alternative patterns.

```
:%s/\v(hello|hola) vim/\1 friend/g
```

The result:

```
hello friend
hola friend
salve vim
bonjour vim
```

Here is the breakdown:

- `%s` runs the substitute command on each line in a file.
- `(hello|hola)` matches *either* "hello" or "hola" and consider it as a group.
- `vim` is the literal word "vim".
- `\1` is the first group, which is either the text "hello" or "hola".
- `friend` is the literal word "friend".

Substituting The Start And The End Of A Pattern

Recall that you can use `\zs` and `\ze` to define the start and the end of a match. This technique works in substitution too. If you have:

```
chocolate pancake
strawberry sweetcake
blueberry hotcake
```

To substitute the "cake" in "hotcake" with "dog" to get a "hotdog":

```
:%s/hot\zscake/dog/g
```

Result:

```
chocolate pancake
strawberry sweetcake
blueberry hotdog
```

Greedy And Non-Greedy

You can substitute the nth match in a line with this trick:

```
One Mississippi, two Mississippi, three Mississippi, four Mississippi, five Mississi\
ppi.
```

To substitute the third "Mississippi" with "Arkansas", run:

```
:s/\v(.{-}\zsMississippi){3}/Arkansas/g
```

The breakdown:

- `:s/` the substitute command.
- `\v` is the magic keyword so you don't have to escape special keywords.
- `.` matches any single character.
- `{-}` performs non-greedy match of 0 or more of the preceding atom.
- `\zsMississippi` makes "Mississippi" the start of the match.
- `(...){3}` looks for the third match.

You have seen the `{3}` syntax earlier in this chapter. In this case, `{3}` will match exactly the third match. The new trick here is `{-}`. It is a non-greedy match. It finds the shortest match of the given pattern. In this case, `(.{-}Mississippi)` matches the least amount of "Mississippi" preceded by any character. Contrast this with `(.*Mississippi)` where it finds the longest match of the given pattern.

If you use `(.{-}Mississippi)`, you get five matches: "One Mississippi", "Two Mississippi", etc. If you use `(.*Mississippi)`, you get one match: the last "Mississippi". `*` is a greedy matcher and `{-}` is a non-greedy matcher. To learn more check out `:h /\{-` and `:h non-greedy`.

Let's do a simpler example. If you have the string:

```
abc1de1
```

You can match "abc1de1" (greedy) with:

```
/a.*1
```

You can match "abc1" (non-greedy) with:

```
/a.\{-}1
```

So if you need to uppercase the longest match (greedy), run:

```
:s/a.*1/\U&/g
```

To get:

```
ABC1DEFG1
```

If you need to uppercase the shortest match (non-greedy), run:

```
:s/a.\{-}1/\U&/g
```

To get:

```
ABC1defg1
```

If you're new to greedy vs non-greedy concept, it can get hard to wrap your head around it. Experiment around with different combinations until you understand it.

Substituting Across Multiple Files

Finally, let's learn how to substitute phrases across multiple files. For this section, assume that you have two files: `food.txt` and `animal.txt`.

Inside `food.txt`:

```
corndog
hotdog
chilidog
```

Inside `animal.txt`:

```
large dog
medium dog
small dog
```

Assume your directory structure looks like this:

```
- food.txt
- animal.txt
```

First, capture both `food.txt` and `animal.txt` inside `:args`. Recall from earlier chapters that `:args` can be used to create a list of file names. There are several ways to do this from inside Vim, one of them is by running this from inside Vim:

```
:args *.txt                captures all txt files in current location
```

To test it, when you run `:args`, you should see:

```
[food.txt] animal.txt
```

Now that all the relevant files are stored inside the argument list, you can perform a multi-file substitution with the `:argdo` command. Run:

```
:argdo %s/dog/chicken/
```

This performs substitution against the all files inside the `:args` list. Finally, save the changed files with:

```
:argdo update
```

`:args` and `:argdo` are useful tools to apply command line commands across multiple files. Try it with other commands!

Substituting Across Multiple Files With Macros

Alternatively, you can also run the substitute command across multiple files with macros. Run:

```
:args *.txt
qq
:%s/dog/chicken/g
:wnext
q
99@q
```

The breakdown:

- `:args *.txt` adds all text files into the `:args` list.
- `qq` starts the macro in the "q" register.
- `:%s/dog/chicken/g` substitutes "dog" with "chicken" on all lines in the current file.
- `:wnext` saves the file then go to the next file on the `args` list.
- `q` stops the macro recording.
- `99@q` executes the macro ninety-nine times. Vim will stop the macro execution after it encounters the first error, so Vim won't actually execute the macro ninety-nine times.

Learning Search And Substitution The Smart Way

The ability to do search well is a necessary skill in editing. Mastering the search lets you to utilize the flexibility of regular expressions to search for any pattern in a file. Take your time to learn these. To get better with regular expression you need to be actively using regular expressions. I once read a book about regular expression without actually doing it and I forgot almost everything I read afterwards. Active coding is the best way to master any skill.

A good way to improve your pattern matching skill is whenever you need to search for a pattern (like "hello 123"), instead of querying for the literal search term (`/hello 123`), try to come up with a pattern for it (something like `/\v(\l+) (\d+)`). Many of these regular expression concepts are also applicable in general programming, not only when using Vim.

Now that you learned about advanced search and substitution in Vim, let's learn one of the most versatile commands, the global command.

Ch13. The Global Command

So far you have learned how to repeat the last change with the dot command (`.`), to replay actions with macros (`q`), and to store texts in the registers (`"`).

In this chapter, you will learn how to repeat a command-line command with the global command.

Global Command Overview

Vim's global command is used to run a command-line command on multiple lines simultaneously.

By the way, you may have heard of the term "Ex Commands" before. In this guide, I refer them as command-line commands. Both Ex commands and command-line commands are the same. They are the commands that start with a colon (`:`). The substitute command in the last chapter was an example of an Ex command. They are called Ex because they originally came from the Ex text editor. I will continue to refer to them as command-line commands in this guide. For a full list of Ex commands, check out `:h ex-cmd-index`.

The global command has the following syntax:

```
:g/pattern/command
```

The `pattern` matches all lines containing that pattern, similar to the pattern in the substitute command. The `command` can be any command-line command. The global command works by executing `command` against each line that matches the `pattern`.

If you have the following expressions:

```
const one = 1;
console.log("one: ", one);

const two = 2;
console.log("two: ", two);

const three = 3;
console.log("three: ", three);
```

To remove all lines containing "console", you can run:

```
:g/console/d
```

Result:

```
const one = 1;

const two = 2;

const three = 3;
```

The global command executes the delete command (`d`) on all lines that match the "console" pattern.

When running the `g` command, Vim makes two scans across the file. On the first run, it scans each line and marks the line that matches the `/console/` pattern. Once all the matching lines are marked, it goes for the second time and executes the `d` command on the marked lines.

If you want to delete all lines containing "const" instead, run:

```
:g/const/d
```

Result:

```
console.log("one: ", one);

console.log("two: ", two);

console.log("three: ", three);
```

Inverse Match

To run the global command on non-matching lines, you can run:

```
:g!/{pattern}/{command}
```

or

```
:v/{pattern}/{command}
```

If you run `:v/console/d`, it will delete all lines *not* containing "console".

Pattern

The global command uses the same pattern system as the substitute command, so this section will serve as a refresher. Feel free to skip to the next section or read along!

If you have these expressions:

```
const one = 1;
console.log("one: ", one);

const two = 2;
console.log("two: ", two);

const three = 3;
console.log("three: ", three);
```

To delete the lines containing either “one” or “two”, run:

```
:g/one\|two/d
```

To delete the lines containing any single digits, run either:

```
:g/[0-9]/d
```

or

```
:g/\d/d
```

If you have the expression:

```
const oneMillion = 1000000;
const oneThousand = 1000;
const one = 1;
```

To match the lines containing between three to six zeroes, run:

```
:g/0\{3,6\}/d
```

Passing A Range

You can pass a range before the `g` command. Here are some ways you can do it:

- `:1,5g/console/d` matches the string “console” between lines 1 and 5 and deletes them.
- `:,5g/console/d` if there is no address before the comma, then it starts from the current line. It looks for the string “console” between the current line and line 5 and deletes them.
- `:3,g/console/d` if there is no address after the comma, then it ends at the current line. It looks for the string “console” between line 3 and the current line and deletes them.

- `:3g/console/d` if you only pass one address without a comma, it executes the command only on line 3. It looks on line 3 and deletes it if has the string "console".

In addition to numbers, you can also use these symbols as range:

- `.` means the current line. A range of `.,3` means between the current line and line 3.
- `$` means the last line in the file. `3,$` range means between line 3 and the last line.
- `+n` means n lines after the current line. You can use it with `.` or without. `3,+1` or `3,.+1` means between line 3 and the line after the current line.

If you don't give it any range, by default it affects the entire file. This is actually not the norm. Most of Vim's command-line commands run on only the current line if you don't pass it any range. The two notable exceptions are the global (`:g`) and the save (`:w`) commands.

Normal Command

You can run a normal command with the global command with `:normal` command-line command.

If you have this text:

```
const one = 1
console.log("one: ", one)

const two = 2
console.log("two: ", two)

const three = 3
console.log("three: ", three)
```

To add a ";" to the end of each line, run:

```
:g/./normal A;
```

Let's break it down:

- `:g` is the global command.
- `/./` is a pattern for "non-empty lines". It matches the lines with at least one character, so it matches the lines with "const" and "console" and it does not match empty lines.
- `normal A;` runs the `:normal` command-line command. `A;` is the normal mode command to insert a ";" at the end of the line.

Executing A Macro

You can also execute a macro with the global command. A macro can be executed with the `normal` command. If you have the expressions:

```
const one = 1
console.log("one: ", one);

const two = 2
console.log("two: ", two);

const three = 3
console.log("three: ", three);
```

Notice that the lines with “const” do not have semi-colons. Let’s create a macro to add a comma to the end of those lines in the register a:

```
qa0A;<Esc>q
```

If you need a refresher, check out the chapter on macro. Now run:

```
:g/const/normal @a
```

Now all lines with “const” will have a “;” at the end.

```
const one = 1;
console.log("one: ", one);

const two = 2;
console.log("two: ", two);

const three = 3;
console.log("three: ", three);
```

If you followed this step-by-step, you will have two semi-colons on the first line. To avoid that, run the global command on line two onward, `:2,$g/const/normal @a`.

Recursive Global Command

The global command itself is a type of a command-line command, so you can technically run the global command inside a global command.

Given the following expressions, if you want to delete the second `console.log` statement:

```
const one = 1;
console.log("one: ", one);

const two = 2;
console.log("two: ", two);

const three = 3;
console.log("three: ", three);
```

If you run:

```
:g/console/g/two/d
```

First, `g` will look for the lines containing the pattern "console" and will find `3` matches. Then the second `g` will look for the line containing the pattern "two" from those three matches. Finally, it will delete that match.

You can also combine `g` with `v` to find positive and negative patterns. For example:

```
:g/console/v/two/d
```

Instead of looking for the line containing the pattern "two", it will look for the lines *not* containing the pattern "two".

Changing The Delimiter

You can change the global command's delimiter like the substitute command. The rules are the same: you can use any single byte character except for alphabets, numbers, `"`, `|`, and `\`.

To delete the lines containing "console":

```
:g@console@d
```

If you are using the substitute command with the global command, you can have two different delimiters:

```
g@one@s+const+let+g
```

Here the global command will look for all lines containing "one". The substitute command will substitute, from those matches, the string "const" with "let".

The Default Command

What happens if you don't specify any command-line command in the global command?

The global command will use the print (`:p`) command to print the current line's text. If you run:

```
:g/console
```

It will print at the bottom of the screen all the lines containing "console".

By the way, here is one interesting fact. Because the default command used by the global command is `p`, this makes the `g` syntax to be:

```
:g/re/p
```

- `g` = the global command
- `re` = the regex pattern
- `p` = the print command

It spells *"grep"*, the same `grep` from the command line. This is **not** a coincidence. The `g/re/p` command originally came from the Ed Editor, one of the original line text editors. The `grep` command got its name from Ed.

Your computer probably still has the Ed editor. Run `ed` from the terminal (hint: to quit, type `q`).

Reversing The Entire Buffer

To reverse the entire file, run:

```
:g/^/m 0
```

`^` is a pattern for the beginning of a line. Use `^` to match all lines, including empty lines.

If you need to reverse only a few lines, pass it a range. To reverse the lines between line five to line ten, run:

```
:5,10g/^/m 0
```

To learn more about the move command, check out `:h :move`.

Aggregating All TODOs

When coding, sometimes I would write TODOs in the file I'm editing:

```
const one = 1;
console.log("one: ", one);
// TODO: feed the puppy

const two = 2;
// TODO: feed the puppy automatically
console.log("two: ", two);

const three = 3;
console.log("three: ", three);
// TODO: create a startup selling an automatic puppy feeder
```

It can be hard to keep track of all the created TODOs. Vim has a `:t` (copy) method to copy all matches to an address. To learn more about the copy method, check out `:h :copy`.

To copy all TODOs to the end of the file for easier introspection, run:

```
:g/TODO/t $
```

Result:

```
const one = 1;
console.log("one: ", one);
// TODO: feed the puppy

const two = 2;
// TODO: feed the puppy automatically
console.log("two: ", two);

const three = 3;
console.log("three: ", three);
// TODO: create a startup selling an automatic puppy feeder

// TODO: feed the puppy
// TODO: feed the puppy automatically
// TODO: create a startup selling an automatic puppy feeder
```

Now I can review all the TODOs I created, find a time to do them or delegate them to someone else, and continue to work on my next task.

If instead of copying them you want to move all the TODOs to the end, use the move command, `:m`:

```
:g/TODO/m $
```

Result:

```
const one = 1;
console.log("one: ", one);

const two = 2;
console.log("two: ", two);

const three = 3;
console.log("three: ", three);

// TODO: feed the puppy
// TODO: feed the puppy automatically
// TODO: create a startup selling an automatic puppy feeder
```

Black Hole Delete

Recall from the register chapter that deleted texts are stored inside the numbered registers (granted they are sufficiently large). Whenever you run `:g/console/d`, Vim stores the deleted lines in the numbered registers. If you delete many lines, you can quickly fill up all the numbered registers. To avoid this, you can always use the black hole register (`"_`) to *not* store your deleted lines into the registers. Run:

```
:g/console/d _
```

By passing _ after `d`, Vim won't use up your scratch registers.

Reduce Multiple Empty Lines To One Empty Line

If you have a text with multiple empty lines:

```
const one = 1;
console.log("one: ", one);


const two = 2;
console.log("two: ", two);

const three = 3;
console.log("three: ", three);
```

You can quickly reduce the empty lines into one empty line with:

```
:g/^$/,/./-1j
```

Result:

```
const one = 1;
console.log("one: ", one);

const two = 2;
console.log("two: ", two);

const three = 3;
console.log("three: ", three);
```

Normally the global command accepts the following form: `:g/pattern/command`. However, you can also run the global command with the following form: `:g/pattern1/,/pattern2/command`. With this, Vim will apply the `command` within `pattern1` and `pattern2`.

With that in mind, let's break down the command `:g/^$/,/./-1j` according to `:g/pattern1/,/pattern2/command`:

- `/pattern1/` is `/^$/`. It represents an empty line (a line with zero character).
- `/pattern2/` is `/./` with `-1` line modifier. `/./` represents a non-empty line (a line with at least one character). The `-1` means the line above that.
- `command` is `j`, the join command (`:j`). In this context, this global command joins all the given lines.

By the way, if you want to reduce multiple empty lines to no lines, run this instead:

```
:g/^$/,/./j
```

A simpler alternative:

```
:g/^$/-j
```

Your text is now reduced to:

```
const one = 1;
console.log("one: ", one);
const two = 2;
console.log("two: ", two);
const three = 3;
console.log("three: ", three);
```

Advanced Sort

Vim has a `:sort` command to sort the lines within a range. For example:

```
d
b
a
e
c
```

You can sort them by running `:sort`. If you give it a range, it will sort only the lines within that range. For example, `:3,5sort` only sorts lines three and five.

If you have the following expressions:

```
const arrayB = [
  "i",
  "g",
  "h",
  "b",
  "f",
  "d",
  "e",
  "c",
  "a",
]

const arrayA = [
  "h",
  "b",
  "f",
```

```
  "d",
  "e",
  "a",
  "c",
]
```

If you need to sort the elements inside the arrays, but not the arrays themselves, you can run this:

```
:g/\[/+1,/\]/-1sort
```

Result:

```
const arrayB = [
  "a",
  "b",
  "c",
  "d",
  "e",
  "f",
  "g",
  "h",
  "i",
]

const arrayA = [
  "a"
  "b",
  "c",
  "d",
  "e",
  "f",
  "h",
]
```

:g/[/+1,/]/-1sort
This is great! But the command looks complicated. Let's break it down. This command also follows the form `:g/pattern1/,/pattern2/command`.
`:g/\[/` is the global command pattern.

- `/\[/+1` is the first pattern. It matches a literal left square bracket "[". The `+1` refers to the line below it.
- `/\]/-1` is the second pattern. It matches a literal right square bracket "]". The `-1` refers to the line above it.
- `/\[/+1,/\]/-1` then refers to any lines between "[" and "]".
- `sort` is a command-line command to sort.

Learn The Global Command The Smart Way

The global command executes the command-line command against all matching lines. With it, you only need to run a command once and Vim will do the rest for you. To become proficient at the global command, two things are required: a good vocabulary of command-line commands and a knowledge of regular expressions. As you spend more time using Vim, you will naturally learn more command-line commands. A regular expression knowledge will require a more active approach. But once you become comfortable with regular expressions, you will be ahead of many.

Some of the examples here are complicated. Do not be intimidated. Really take your time to understand them. Learn to read the patterns. Do not give up.

Whenever you need to run multiple commands, pause and see if you can use the `g` command. Identify the best command for the job and write a pattern to target as many things at once.

Now that you know how powerful the global command is, let's learn how to use the external commands to increase your tool arsenals.

Ch14. External Commands

Inside the Unix system, you will find many small, hyper-specialized commands that does one thing (and does it well). You can chain these commands to work together to solve a complex problem. Wouldn't it be great if you can use these commands from inside Vim?

Definitely. In this chapter, you will learn how extend Vim to work seamlessly with external commands.

The Bang Command

Vim has a bang (`!`) command that can do three things:

1. Read the STDOUT of an external command into the current buffer.
2. Write the content of your buffer as the STDIN to an external command.
3. Execute an external command from inside Vim.

Let's go through each of them.

Reading The STDOUT Of A Command Into Vim

The syntax to read the STDOUT of an external command into the current buffer is:

```
:r !{cmd}
```

`:r` is Vim's read command. If you use it without `!`, you can use it to get the content of a file. If you have a file `file1.txt` in the current directory and you run:

```
:r file1.txt
```

Vim will put the content of `file1.txt` into the current buffer.

If you run the `:r` command followed by a `!` and an external command, the output of that commmand will be inserted into the current buffer. To get the result of the `ls` command, run:

```
:r !ls
```

It returns something like:

```
file1.txt
file2.txt
file3.txt
```

You can read the data from the `curl` command:

```
:r !curl -s 'https://jsonplaceholder.typicode.com/todos/1'
```

The `r` command also accepts an address:

```
:10r !cat file1.txt
```

Now the STDOUT from running `cat file1.txt` will be inserted after line 10.

Writing The Buffer Content Into An External Command

The command `:w`, in addition to saving a file, can be used to pass the text in the current buffer as the STDIN for an external command. The syntax is:

```
:w !{cmd}
```

If you have these expressions:

```
console.log("Hello Vim");
console.log("Vim is awesome");
```

Make sure you have node[27] installed in your machine, then run:

```
:w !node
```

Vim will use `node` to execute the JavaScript expressions to print "Hello Vim" and "Vim is awesome".

When using the `:w` command, Vim uses all texts in the current buffer, similar to the global command (most command-line commands, if you don't pass it a range, only executes the command against the current line). If you pass `:w` a specific address:

[27] https://nodejs.org/en/

```
:2w !node
```

Vim only uses the text from the second line into the `node` interpreter.

There is a subtle but significant difference between `:w !node` and `:w! node`. With `:w !node`, you are "writing" the text in the current buffer into the external command `node`. With `:w! node`, you are force-saving a file and naming the file "node".

Executing An External Command

You can execute an external command from inside Vim with the bang command. The syntax is:

```
:!cmd
```

To see the content of the current directory in the long format, run:

```
:!ls -ls
```

To kill a process that is running on PID 3456, you can run:

```
:!kill -9 3456
```

You can run any external command without leaving Vim so you can stay focused on your task.

Filtering Texts

If you give `!` a range, it can be used to filter texts. Suppose you have the following texts:

```
hello vim
hello vim
```

Let's uppercase the current line using the `tr` (translate) command. Run:

```
:.!tr '[:lower:]' '[:upper:]'
```

The result:

```
HELLO VIM
hello vim
```

The breakdown:

- `.!` executes the filter command on the current line.
- `!tr '[:lower:]' '[:upper:]'` calls the `tr` command to replace all lowercase characters with uppercase ones.

It is imperative to pass a range to run the external command as a filter. If you try running the command above without the `.` (`:!tr '[:lower:]' '[:upper:]'`), you will see an error.

Let's assume that you need to remove the second column on both lines with the `awk` command:

```
:%!awk "{print $1}"
```

The result:

```
hello
hello
```

The breakdown:

- `:%!` executes the filter command on all lines (`%`).
- `awk "{print $1}"` prints only the first column of the match.

You can chain multiple commands with the chain operator (`|`) just like in the terminal. Let's say you have a file with these delicious breakfast items:

```
name price
chocolate pancake 10
buttermilk pancake 9
blueberry pancake 12
```

If you need to sort them based on the price and display only the menu with an even spacing, you can run:

```
:%!awk 'NR > 1' | sort -nk 3 | column -t
```

The result:

```
buttermilk pancake 9
chocolate pancake 10
blueberry pancake 12
```

The breakdown:

- `:%!` applies the filter to all lines (`%`).
- `awk 'NR > 1'` displays the texts only from row number two onwards.
- `|` chains the next command.
- `sort -nk 3` sorts numerically (`n`) using the values from column 3 (`k 3`).
- `column -t` organizes the text with even spacing.

Normal Mode Command

Vim has a filter operator (`!`) in the normal mode. If you have the following greetings:

```
hello vim
hola vim
bonjour vim
salve vim
```

To uppercase the current line and the line below, you can run:

```
!jtr '[a-z]' '[A-Z]'
```

The breakdown:

- `!j` runs the normal command filter operator (`!`) targetting the current line and the line below it. Recall that because it is a normal mode operator, the grammar rule `verb + noun` applies. `!` is the verb and `j` is the noun.
- `tr '[a-z]' '[A-Z]'` replaces the lowercase letters with the uppercase letters.

The filter normal command only works on motions / text objects that are at least one line or longer. If you had tried running `!iwtr '[a-z]' '[A-Z]'` (execute `tr` on inner word), you will find that it applies the `tr` command on the entire line, not the word your cursor is on.

Learn External Commands The Smart Way

Vim is not an IDE. It is a lightweight modal editor that is highly extensible by design. Because of this extensibility, you have an easy access to any external command in your system. Armed with these external commands, Vim is one step closer from becoming an IDE. Someone said that the Unix system is the first IDE ever.

The bang command is as useful as how many external commands you know. Don't worry if your external command knowledge is limited. I still have a lot to learn too. Take this as a motivation for continuous learning. Whenever you need to modify a text, look if there is an external command that can solve your problem. Don't worry about mastering everything, just learn the ones you need to complete the current task.

Ch15. Command-line Mode

In the last three chapters, you learned how to use the search commands (`/`, `?`), substitute command (`:s`), global command (`:g`), and external command (`!`). These are examples of command-line mode commands.

In this chapter, you will learn various tips and tricks for the command-line mode.

Entering And Exiting The Command-line Mode

The command-line mode is a mode in itself, just like normal mode, insert mode, and visual mode. When you are in this mode, the cursor goes to the bottom of the screen where you can type in different commands.

There are 4 different commands you can use to enter the command-line mode:

- Search patterns (`/`, `?`)
- Command-line commands (`:`)
- External commands (`!`)

You can enter the command-line mode from the normal mode or the visual mode.

To leave the command-line mode, you can use `<Esc>`, `Ctrl-c, or Ctrl-[`.

Other literatures might refer the "Command-line command" as "Ex command" and the "External command" as "filter command" or "bang operator".

Repeating The Previous Command

You can repeat the previous command-line command or external command with `@:`.

If you just ran `:s/foo/bar/g`, running `@:` repeats that substitution. If you just ran `:.!tr '[a-z]' '[A-Z]'`, running `@:` repeats the last external command translation filter.

Command-line Mode Shortcuts

While in the command-line mode, you can move to the left or to the right, one character at a time, with the `Left` or `Right` arrow.

If you need to move word-wise, use `Shift-Left` or `Shift-Right` (in some OS, you might have to use `Ctrl` instead of `Shift`).

To go to the start of the line, use `Ctrl-b`. To go to the end of the line, use `Ctrl-e`.

Similar to the insert mode, inside the command-line mode, you have three ways to delete characters:

```
Ctrl-H    Delete one character
Ctrl-W    Delete one word
Ctrl-U    Delete the entire line
```

Finally, if you want to edit the command like you would a normal textfile use `Ctrl-f`.

This also allows you to search through the previous commands, edit them and rerun them by pressing `<Enter>` in "command-line editing normal mode".

Register And Autocomplete

While in the command-line mode, you can insert texts from Vim register with `Ctrl-R` the same way as the insert mode. If you have the string "foo" saved in the register a, you can insert it by running `Ctrl-R a`. Everything that you can get from the register in the insert mode, you can do the same from the command-line mode.

In addition, you can also get the word under the cursor with `Ctrl-R Ctrl-W` (`Ctrl-R Ctrl-A` for the WORD under cursor). To get the line under the cursor, use `Ctrl-R Ctrl-L`. To get the filename under the cursor, use `Ctrl-R Ctrl-F`.

You can also autocomplete existing commands. To autocomplete the `echo` command, while in the command-line mode, type "ec", then press `<Tab>`. You should see on the bottom left Vim commands starting with "ec" (example: `echo echoerr echohl echomsg econ`). To go to the next option, press either `<Tab>` or `Ctrl-N`. To go the previous option, press either `<Shift-Tab>` or `Ctrl-P`.

Some command-line commands accept file names as arguments. One example is `edit`. You can autocomplete here too. After typing the command, `:e ` (don't forget the space), press `<Tab>`. Vim will list all the relevant file names that you can choose from so you don't have to type it from scratch.

History Window And Command-line Window

You can view the histoy of command-line commands and search terms (this requires the `+cmdline_hist` feature).

To open the command-line history, run `:his :`. You should see something like the following:

```
##  cmd History
2  e file1.txt
3  g/foo/d
4  s/foo/bar/g
```

Vim lists the history of all the `:` commands you run. By default, Vim stores the last 50 commands. To change the amount of the entries that Vim remembers to 100, you run `set history=100`.

A more useful use of the command-line history is through the command-line window,`q:`. This will open a searchable, editable history window. Suppose you have these expressions in the history when you press `q:`:

```
51  s/verylongsubstitutionpattern/pancake/g
52  his :
53  wq
```

If your current task is to do `s/verylongsubstitutionpattern/donut/g`, instead of typing the command from scratch, why don't you reuse `s/verylongsubstitutionpattern/pancake/g`? After all, the only thing that's different is the word substitute, "donut" vs "pancake". Everything else is the same.

After you ran `q:`, find that `s/verylongsubstitutionpattern/pancake/g` in the history (you can use the Vim navigation in this environment) and edit it directly! Change "pancake" to "donut" inside the history window, then press `<Enter>`. Boom! Vim executes `s/verylongsubstitutionpattern/donut/g` for you. Super convenient!

Similarly, to view the search history, run `:his /` or `:his ?`. To open the search history window where you can search and edit past history, run `q/` or `q?`.

To quit this window, press `Ctrl-C`, `Ctrl-W C`, or type `:quit`.

More Command-line Commands

Vim has hundreds of built-in commands. To see all the commands Vim have, check out `:h ex-cmd-index` or `:h :index`.

Learn Command-line Mode The Smart Way

Compared to the other three modes, the command-line mode is like the Swiss Army knife of text editing. You can edit text, modify files, and execute commands, just to name a few. This chapter is a collection of odds and ends of the command-line mode. It also brings Vim modes into closure. Now that you know how to use the normal, insert, visual, and command-line mode you can edit text with Vim faster than ever.

It's time to move away from Vim modes and learn how to do an even faster navigation with Vim tags.

Ch16. Tags

One useful feature in text editing is being able to go to any definition quickly. In this chapter, you will learn how to use Vim tags to do that.

Tag Overview

Suppose someone handed you a new codebase:

```
one = One.new
one.donut
```

`One`? `donut`? Well, these might have been obvious to the developers writing the code way back then, but now those developers are no longer here and it is up to you to understand these obscure codes. One way to help understand this is to follow the source code where `One` and `donut` are defined.

You can search for them with either `fzf` or `grep` (or `vimgrep`), but in this case, tags are faster.

Think of tags like an address book:

```
Name    Address
Iggy1   1234 Cool St, 11111
Iggy2   9876 Awesome Ave, 2222
```

Instead of having a name-address pair, tags store definitions paired with addresses.

Let's assume that you have these two Ruby files inside the same directory:

```ruby
## one.rb
class One
  def initialize
    puts "Initialized"
  end

  def donut
    puts "Bar"
  end
end
```

and

```
## two.rb
require './one'

one = One.new
one.donut
```

To jump to a definition, you can use `Ctrl-]` in the normal mode. Inside `two.rb`, go to the line where `one.donut` is and move the cursor over `donut`. Press `Ctrl-]`.

Whoops, Vim could not find the tag file. You need to generate the tag file first.

Tag Generator

Modern Vim does not come with tag generator, so you will have to download an external tag generator. There are several options to choose:

- ctags = C only. Available almost everywhere.
- exuberant ctags = One of the most popular ones. Has many language support.
- universal ctags = Similar to exuberant ctags, but newer.
- etags = For Emacs. Hmm…
- JTags = Java
- ptags.py = Python
- ptags = Perl
- gnatxref = Ada

If you look at Vim tutorials online, many will recommend exuberant ctags[28]. It supports 41 programming languages[29]. I used it and it worked great. However, because it has not been maintained since 2009, Universal ctags would be a better choice. It works similar to exuberant ctags and is currently being maintained.

I won't go into details on how to install the universal ctags. Check out the universal ctags[30] repository for more instructions.

Assuming you have the universal ctags installed, let's generate a basic tag file. Run:

```
ctags -R .
```

The `R` option tells ctags to run a recursive scan from your current location (`.`). You should see a `tags` file in your current directory. Inside you will see something like this:

[28] http://ctags.sourceforge.net/
[29] http://ctags.sourceforge.net/languages.html
[30] https://github.com/universal-ctags/ctags

```
!_TAG_FILE_FORMAT        2        /extended format; --format=1 will not append ;" to lines/
!_TAG_FILE_SORTED        1        /0=unsorted, 1=sorted, 2=foldcase/
!_TAG_OUTPUT_FILESEP        slash        /slash or backslash/
!_TAG_OUTPUT_MODE        u-ctags        /u-ctags or e-ctags/
!_TAG_PATTERN_LENGTH_LIMIT        96        /0 for no limit/
!_TAG_PROGRAM_AUTHOR        Universal Ctags Team        //
!_TAG_PROGRAM_NAME        Universal Ctags        /Derived from Exuberant Ctags/
!_TAG_PROGRAM_URL        <https://ctags.io/>        /official site/
!_TAG_PROGRAM_VERSION        0.0.0        /b43eb39/
One        one.rb        /^class One$/;"        c
donut        one.rb        /^  def donut$/;"        f        class:One
initialize        one.rb        /^  def initialize$/;"        f        class:One
```

Yours might look a little different depending on your Vim setting and the ctags generator. A tag file is composed of two parts: the tag metadata and the tag list. These metadata (`!TAG_FILE...`) are usually controlled by the ctags generator. I won't discuss it here, but feel free to check their docs for more! The tag list is a list of all the definitions indexed by ctags.

Now go to `two.rb`, put the cursor on `donut`, and type `Ctrl-]`. Vim will take you to the file `one.rb` on the line where `def donut` is. Success! But how did Vim do this?

Tags Anatomy

Let's look at the `donut` tag item:

```
donut        one.rb        /^  def donut$/;"        f        class:One
```

The above tag item is composed of four components: a `tagname`, a `tagfile`, a `tagaddress`, and tag options.

- `donut` is the `tagname`. When your cursor is on "donut", Vim searches the tag file for a line that has the "donut" string.
- `one.rb` is the `tagfile`. Vim looks for a file `one.rb`.
- `/^  def donut$/` is the `tagaddress`. `/.../` is a pattern indicator. `^` is a pattern for the first element on a line. It is followed by two spaces, then the string `def donut`. Finally, `$` is a pattern for the last element on a line.
- `f class:One` is the tag option that tells Vim that the function `donut` is a function (`f`) and is part of the `One` class.

Let's look at another item in the tag list:

```
One        one.rb        /^class One$/;"        c
```

This line works the same way as the `donut` pattern:

- `One` is the `tagname`. Note that with tags, the first scan is case sensitive. If you have `One` and `one` on the list, Vim will prioritize `One` over `one`.
- `one.rb` is the `tagfile`. Vim looks for a file `one.rb`.
- `/^class One$/` is the `tagaddress` pattern. Vim looks for a line that starts with (`^`) `class` and ends with (`$`) `One`.
- `c` is one of the possible tag options. Since `One` is a ruby class and not a procedure, it marks it with a `c`.

Depending on which tag generator you use, the content of your tag file may look different. At minimum, a tag file must have either one of these formats:

```
1.  {tagname} {TAB} {tagfile} {TAB} {tagaddress}
2.  {tagname} {TAB} {tagfile} {TAB} {tagaddress} {term} {field} ..
```

The Tag File

You have learned that a new file, `tags`, is created after running `ctags -R .`. How does Vim know where to look for the tag file?

If you run `:set tags?`, you might see `tags=./tags,tags` (depending on your Vim settings, it might be different). Here Vim looks for all tags in the path of the current file in the case of `./tags` and the current directory (your project root) in the case of `tags`.

Also in the case of `./tags`, Vim will first look for a tag file inside the path of your current file regardless how nested it is, then it will look for a tag file of the current directory (project root). Vim stops after it finds the first match.

If your `'tags'` file had said `tags=./tags,tags,/user/iggy/mytags/tags`, then Vim will also look at the `/user/iggy/mytags` directory for a tag file after Vim finishes searching `./tags` and `tags` directory. You don't have to store your tag file inside your project, you can keep them separate.

To add a new tag file location, use the following:

```
set tags+=path/to/my/tags/file
```

Generating Tags For A Large Project

If you tried to run ctags in a large project, it may take a long time because Vim also looks inside every nested directories. If you are a Javascript developer, you know that `node_modules` can be very large. Imagine if you have a five sub-projects and each contains its own `node_modules` directory. If you run `ctags -R .`, ctags will try to scan through all 5 `node_modules`. You probably don't need to run ctags on `node_modules`.

To run ctags excluding the `node_modules`, run:

```
ctags -R --exclude=node_modules .
```

This time it should take less than a second. By the way, you can use the `exclude` option multiple times:

```
ctags -R --exclude=.git --exclude=vendor --exclude=node_modules --exclude=db --exclu\
de=log .
```

The point is, if you want to omit a directory, `--exclude` is your best friend.

Tags Navigation

You can get good mileage using only `Ctrl-]`, but let's learn a few more tricks. The tag jump key `Ctrl-]` has an command-line mode alternative: `:tag {tag-name}`. If you run:

```
:tag donut
```

Vim will jump to the `donut` method, just like doing `Ctrl-]` on "donut" string. You can autocomplete the argument too, with `<Tab>`:

```
:tag d<Tab>
```

Vim lists all tags that starts with "d". In this case, "donut".

In a real project, you may encounter multiple methods with the same name. Let's update the two ruby files from earlier. Inside `one.rb`:

```
## one.rb
class One
  def initialize
    puts "Initialized"
  end

  def donut
    puts "one donut"
  end

  def pancake
    puts "one pancake"
  end
end
```

Inside `two.rb`:

```
## two.rb
require './one.rb'

def pancake
  "Two pancakes"
end

one = One.new
one.donut
puts pancake
```

If you are coding along, don't forget to run `ctags -R .` again since you now have several new procedures. You have two instances of the `pancake` procedure. If you are inside `two.rb` and you pressed `Ctrl-]`, what would happen?

Vim will jump to `def pancake` inside `two.rb`, not the `def pancake` inside `one.rb`. This is because Vim sees the `pancake` procedure inside `two.rb` as having a higher priority than the other `pancake` procedure.

Tag Priority

Not all tags are equal. Some tags have higher priorities. If Vim is presented with duplicate item names, Vim checks the priority of the keyword. The order is:

1. A fully matched static tag in the current file.

2. A fully matched global tag in the current file.
3. A fully matched global tag in a different file.
4. A fully matched static tag in another file.
5. A case-insensitively matched static tag in the current file.
6. A case-insensitively matched global tag in the current file.
7. A case-insensitively matched global tag in the a different file.
8. A case-insensitively matched static tag in the current file.

According to the priority list, Vim prioritizes the exact match found on the same file. That's why Vim chooses the `pancake` procedure inside `two.rb` over the `pancake` procedure inside `one.rb`. There are some exceptions to the priority list above depending on your `'tagcase'`, `'ignorecase'`, and `'smartcase'` settings, but I will not discuss them here. If you are interested, check out `:h tag-priority`.

Selective Tag Jumps

It would be nice if you can choose which tag items to jump to instead of always going to the highest priority tag item. Maybe you actually need to jump to the `pancake` method in `one.rb` and not the one in `two.rb`. To do that, you can use `:tselect`. Run:

```
:tselect pancake
```

You will see, on the bottom of the screen:

```
## pri kind tag               file
1 F C f    pancake           two.rb
             def pancake
2 F   f    pancake           one.rb
             class:One
             def pancake
```

If you type `2`, Vim will jump to the procedure in `one.rb`. If you type `1`, Vim will jump to the procedure in `two.rb`.

Pay attention to the `pri` column. You have `F C` on the first match and `F` on the second match. This is what Vim uses to determine the tag priotity. `F C` means a fully-matched (`F`) global tag in the current (`C`) file. `F` means only a fully-matched (`F`) global tag. `F C` always have a higher priority than `F`.

If you run `:tselect donut`, Vim also prompts you to select which tag item to jump to, even though there is only one option to choose from. Is there a way for Vim to prompt the tag list only if there are multiple matches and to jump immediately if there is only one tag found?

Of course! Vim has a `:tjump` method. Run:

```
:tjump donut
```

Vim will immediately jump to the `donut` procedure in `one.rb`, much like running `:tag donut`. Now run:

```
:tjump pancake
```

Vim will prompt you tag options to choose from, much like running `:tselect pancake`. With `tjump` you get the best of both methods.

Vim has a normal mode key for `tjump`: `g Ctrl-]`. I personally like `g Ctrl-]` better than `Ctrl-]`.

Autocompletion With Tags

Tags can assist autocompletions. Recall from chapter 6, Insert Mode, that you can use `Ctrl-X` sub-mode to do various autocompletions. One autocompletion sub-mode that I did not mention was `Ctrl-]`. If you do `Ctrl-X Ctrl-]` while in the insert mode, Vim will use the tag file for autocompletion.

If you go into the insert mode and type `Ctrl-x Ctrl-]`, you will see:

```
One
donut
initialize
pancake
```

Tag Stack

Vim keeps a list of all the tags you have jumped to and from in a tag stack. You can see this stack with `:tags`. If you had first tag-jumped to `pancake`, followed by `donut`, and run `:tags`, you will see:

```
  # TO tag         FROM line  in file/text
  1  1 pancake            10  ch16_tags/two.rb
  2  1 donut               9  ch16_tags/two.rb
>
```

Note the › symbol above. It shows your current position in the stack. To "pop" the stack to go back to one previous stack, you can run `:pop`. Try it, then run `:tags` again:

```
  # TO tag         FROM line  in file/text
  1  1 pancake            10  puts pancake
> 2  1 donut              9  one.donut
```

Note that the `>` symbol is now on line two, where the `donut` is. `pop` one more time, then run `:tags` again:

```
  # TO tag         FROM line  in file/text
> 1  1 pancake            10  puts pancake
  2  1 donut              9  one.donut
```

In normal mode, you can run `Ctrl-t` to achieve the same effect as `:pop`.

Automatic Tag Generation

One of the biggest drawbacks of Vim tags is that each time you make a significant change, you have to regenerate the tag file. If you recently renamed the `pancake` procedure to the `waffle` procedure, the tag file did not know that the `pancake` procedure had been renamed. It still stored `pancake` in the list of tags. You have to run `ctags -R .` to create an updated tag file. Recreating a new tag file this way can be cumbersome.

Luckily there are several methods you can employ to generate tags automatically.

Generate A Tag On Save

Vim has an autocommand (`autocmd`) method to execute any command on an event trigger. You can use this to generate tags on each save. Run:

```
:autocmd BufWritePost *.rb silent !ctags -R .
```

Breakdown:

- `autocmd` is a command-line command. It accepts an event, file pattern, and a command.
- `BufWritePost` is an event for saving a buffer. Each time you save a file, you trigger a `BufWritePost` event.
- `.rb` is a file pattern for ruby files.
- `silent` is actually part of the command you are passing. Without this, Vim will display `press ENTER or type command to continue` each time you trigger the autocommand.
- `!ctags -R .` is the command to execute. Recall that `!cmd` from inside Vim executes terminal command.

Now each time you save from inside a ruby file, Vim will run `ctags -R .`.

Using Plugins

There are several plugins to generate ctags automatically:

- vim-gutentags[31]
- vim-tags[32]
- vim-easytags[33]
- vim-autotag[34]

I use vim-gutentags. It is simple to use and will work right out of the box.

Ctags And Git Hooks

Tim Pope, author of many great Vim plugins, wrote a blog suggesting to use git hooks. Check it out[35].

Learn Tags The Smart Way

A tag is useful once configured properly. Suppose you are faced with a new codebase and you want to understand what `functionFood` does, you can easily read it by jumping to its definition. Inside it, you learn that it also calls `functionBreakfast`. You follow it and you learn that it calls `functionPancake`. Your function call graph looks something like this:

```
functionFood -> functionBreakfast -> functionPancake
```

This gives you insight that this code flow is related to having a pancake for breakfast.

To learn more about tags, check out `:h tags`. Now that you know how to use tags, let's explore a different feature: folding.

[31] https://github.com/ludovicchabant/vim-gutentags
[32] https://github.com/szw/vim-tags
[33] https://github.com/xolox/vim-easytags
[34] https://github.com/craigemery/vim-autotag
[35] https://tbaggery.com/2011/08/08/effortless-ctags-with-git.html

Ch17. Fold

When you read a file, often there are many irrelevant texts that hinder you from understanding what that file does. To hide the unnecessary noise, use Vim fold.

In this chapter, you will learn different ways to fold a file.

Manual Fold

Imagine that you are folding a sheet of paper to cover some text. The actual text does not go away, it is still there. Vim fold works the same way. It folds a range of text, hiding it from display without actually deleting it.

The fold operator is `z` (when a paper is folded, it is shaped like the letter z).

Suppose you have this text:

```
Fold me
Hold me
```

Type `zfj`. Vim folds both lines into one. You should see something like this:

```
+-- 2 lines: Fold me -----
```

Here is the breakdown:

- `zf` `zf` is the fold operator.
- `j` is the motion for the fold operator.

You can open a folded text with `zo`. To close the fold, use `zc`.

Fold is an operator, so it follows the grammar rule (`verb + noun`). You can pass the fold operator with a motion or text object. To fold an inner paragraph, run `zfip`. To fold to the end of a file, run `zfG`. To fold the texts between `{` and `}`, run `zfa{`.

You can fold from the visual mode. Highlight the area you want to fold (`v`, `V`, or `Ctrl-v`), then run `zf`.

You can execute a fold from the command-line mode with the `:fold` command. To fold the current line and the line after it, run:

```
:,+1fold
```

`,+1` is the range. If you don't pass parameters to the range, it defaults to the current line. `+1` is the range indicator for the next line. To fold the lines 5 to 10, run `:5,10fold`. To fold from the current line to the end of the line, run `:,$fold`.

There are many other fold and unfold commands. I find them too many to remember when starting out. The most useful ones are:

- `zR` to open all folds.
- `zM` to close all folds.
- `za` toggle a fold.

You can run `zR` and `zM` on any line, but `za` only works when you are on a folded / unfolded line. To learn more folding commands, check out `:h fold-commands`.

Different Fold Methods

The section above covers Vim's manual fold. There are six different folding methods in Vim:

1. Manual
2. Indent
3. Expression
4. Syntax
5. Diff
6. Marker

To see which folding method you are currently using, run `:set foldmethod?`. By default, Vim uses the `manual` method.

In the rest of the chapter, you will learn the other five folding methods. Let's get started with the indent fold.

Indent Fold

To use an indent fold, change the `'foldmethod'` to indent:

```
:set foldmethod=indent
```

Suppose that you have the text:

```
One
  Two
  Two again
```

If you run `:set foldmethod=indent`, you will see:

```
One
+-- 2 lines: Two -----
```

With indent fold, Vim looks at how many spaces each line has at the beginning and compares it with the `'shiftwidth'` option to determine its foldability. `'shiftwidth'` returns the number of spaces required for each step of the indent. If you run:

```
:set shiftwidth?
```

Vim's default `'shiftwidth'` value is 2. On the text above, there are two spaces between the start of the line and the text "Two" and "Two again". When Vim sees the number of spaces and that the `'shiftwidth'` value is 2, Vim considers that line to have an indent fold level of one.

Suppose this time you only one space between the start of the line and the text:

```
One
 Two
 Two again
```

Right now if you run `:set foldmethod=indent`, Vim does not fold the indented line because there isn't sufficient space on each line. One space is not considered an indentation. However, if you change the `'shiftwidth'` to 1:

```
:set shiftwidth=1
```

The text is now foldable. It is now considered an indentation.

Restore the `shiftwidth` back to 2 and the spaces between the texts to two again. In addition, add two additional texts:

```
One
  Two
  Two again
    Three
    Three again
```

Run fold (`zM`), you will see:

```
One
+-- 4 lines: Two -----
```

Unfold the folded lines (`zR`), then put your cursor on "Three" and toggle the text's folding state (`za`):

```
One
  Two
  Two again
+-- 2 lines: Three -----
```

What's this? A fold within a fold?

Nested folds are valid. The text "Two" and "Two again" have fold level of one. The text "Three" and "Three again" have fold level of two. If you have a foldable text with a higher fold level within a foldable text, you will have multiple fold layers.

Marker Fold

To use a marker fold, run:

```
:set foldmethod=marker
```

Suppose you have the text:

```
Hello

{{{
world
vim
}}}
```

Run `zM`, you will see:

```
hello

+-- 4 lines: -----
```

Vim sees `{{{` and `}}}` as fold indicators and folds the texts between them. With the marker fold, Vim looks for special markers, defined by `'foldmarker'` option, to mark folding areas. To see what markers Vim uses, run:

```
:set foldmarker?
```

By default, Vim uses `{{{` and `}}}` as indicators. If you want to change the indicator to another texts, like "coffee1" and "coffee2":

```
:set foldmarker=coffee1,coffee2
```

If you have the text:

```
hello

coffee1
world
vim
coffee2
```

Now Vim uses `coffee1` and `coffee2` as the new folding markers. As a side note, an indicator must be a literal string and cannot be a regex.

Syntax Fold

Syntax fold is determined by syntax language highlighting. If you use a language syntax plugin like vim-polyglot[36], the syntax fold will work right out of the box. Just change the fold method to syntax:

```
:set foldmethod=syntax
```

Let's assume you are editing a JavaScript file and you have vim-polyglot installed. If you have an array like the following:

```
const nums = [
  one,
  two,
  three,
  four
]
```

It will be folded with a syntax fold. When you define a syntax highlighting for a particular language (typically inside the `syntax/` directory), you can add a `fold` attribute to make it foldable. Below is a snippet from vim-polyglot JavaScript syntax file. Notice the `fold` keyword at the end.

[36] https://github.com/sheerun/vim-polyglot

```
syntax region  jsBracket                        matchgroup=jsBrackets           start\
=/\[/ end=/\]/ contains=@jsExpression,jsSpreadExpression extend fold
```

This guide won't cover the `syntax` feature. If you're curious, check out `:h syntax.txt`.

Expression Fold

Expression fold allows you to define an expression to match for a fold. After you define the fold expressions, Vim scans each line for the value of `'foldexpr'`. This is the variable that you have to configure to return the appropriate value. If the `'foldexpr'` returns `0`, then the line is not folded. If it returns `1`, then that line has a fold level of `1`. If it returns `2`, then that line has a fold level of `2`. There are more values other than integers, but I won't go over them. If you are curious, check out `:h fold-expr`.

First, let's change the foldmethod:

```
:set foldmethod=expr
```

Suppose you have a list of breakfast foods and you want to fold all breakfast items starting with "p":

```
donut
pancake
pop-tarts
protein bar
salmon
scrambled eggs
```

Next, change the `foldexpr` to capture the expressions starting with "p":

```
:set foldexpr=getline(v:lnum)[0]==\\"p\\"
```

The expression above looks complicated. Let's break it down:

- `:set foldexpr` sets up the `'foldexpr'` option to accept a custom expression.
- `getline()` is a Vimscript function that returns the content of any given line. If you run `:echo getline(5)`, it will return the content of line `5`.
- `v:lnum` is Vim's special variable for the `'foldexpr'` expression. Vim scans each line and at that moment stores each line's number in `v:lnum` variable. On line `5`, `v:lnum` has value of `5`. On line `10`, `v:lnum` has value of `10`.

- `[0]` in the context of `getline(v:lnum)[0]` is the first character of each line. When Vim scans a line, `getline(v:lnum)` returns the content of each line. `getline(v:lnum)[0]` returns the first character of each line. On the first line of our list, "donut", `getline(v:lnum)[0]` returns "d". On the second line of our list, "pancake", `getline(v:lnum)[0]` returns "p".
- `==\\"p\\"` is the second half of the equality expression. It checks if the expression you just evaluated is equal to "p". If it is true, it returns 1. If it is false, it returns 0. In Vim, 1 is truthy and 0 is falsy. So on the lines that start with an "p", it returns 1. Recall if a `'foldexpr'` has a value of 1, then it has a fold level of 1.

After running this expression, you should see:

```
donut
+-- 3 lines: pancake -----
salmon
scrambled eggs
```

Diff Fold

Vim can do a diff procedure to compare two or more files.

If you have `file1.txt`:

```
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
```

And `file2.txt`:

```
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
emacs is ok
```

Run `vimdiff file1.txt file2.txt`:

```
+-- 3 lines: vim is awesome -----
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
vim is awesome
[vim is awesome] / [emacs is ok]
```

Vim automatically folds some of the identical lines. When you are running the `vimdiff` command, Vim automatically uses `foldmethod=diff`. If you run `:set foldmethod?`, it will return `diff`.

Persisting Fold

You loses all fold information when you close the Vim session. If you have this file, `count.txt`:

```
one
two
three
four
five
```

Then do a manual fold from line "three" down (`:3,$fold`):

```
one
two
+-- 3 lines: three ---
```

When you exit Vim and reopen `count.txt`, the folds are no longer there!

To preserve the folds, after folding, run:

```
:mkview
```

Then when you open up `count.txt`, run:

```
:loadview
```

Your folds are restored. However, you have to manually run `mkview` and `loadview`. I know that one of these days, I will forget to run `mkview` before closing the file and I will lose all the folds. How can we automate this process?

To automatically run `mkview` when you close a `.txt` file and run `loadview` when you open a `.txt` file, add this in your vimrc:

```
autocmd BufWinLeave *.txt mkview
autocmd BufWinEnter *.txt silent loadview
```

Recall that `autocmd` is used to execute a command on an event trigger. The two events here are:

- `BufWinLeave` for when you remove a buffer from a window.
- `BufWinEnter` for when you load a buffer in a window.

Now after you fold inside a `.txt` file and exit Vim, the next time you open that file, your fold information will be restored.

By default, Vim saves the fold information when running `mkview` inside `~/.vim/view` for the Unix system. For more information, check out `:h 'viewdir'`.

Learn Fold The Smart Way

When I first started Vim, I neglected ot learn fold because I didn't think it was useful. However, the longer I code, the more useful I find folding is. Strategically placed folds can give you a better overview of the text structure, like a book's table of content.

When you learn fold, start with the manual fold because that can be used on-the-go. Then gradually learn different tricks to do indent and marker folds. Finally, learn how to do syntax and expression folds. You can even use the latter two to write your own Vim plugins.

Ch18. Git

Vim and git are two great tools for two different things. Git is a version control tool. Vim is a text editor.

In this chapter, you will learn different ways to integrate Vim and git together.

Diffing

Recall in the previous chapter, you can run a `vimdiff` command to show the differences between multiple files.

Suppose you have two files, `file1.txt` and `file2.txt`.

Inside `file1.txt`:

```
pancakes
waffles
apples

milk
apple juice

yogurt
```

Inside `file2.txt`:

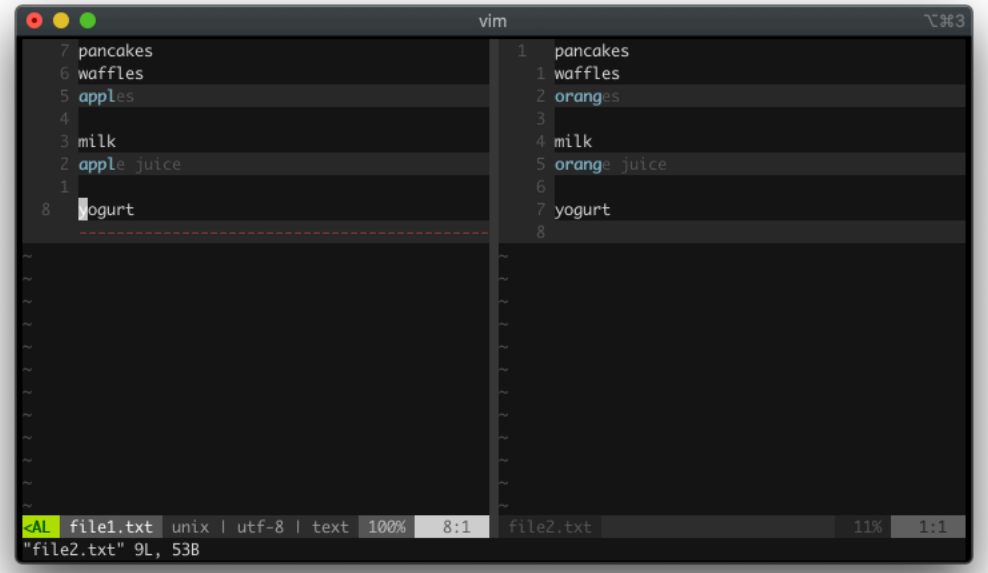
```
pancakes
waffles
oranges

milk
orange juice

yogurt
```

To see the differences between both files, run:

```
vimdiff file1.txt file2.txt
```

Alternatively you could run:

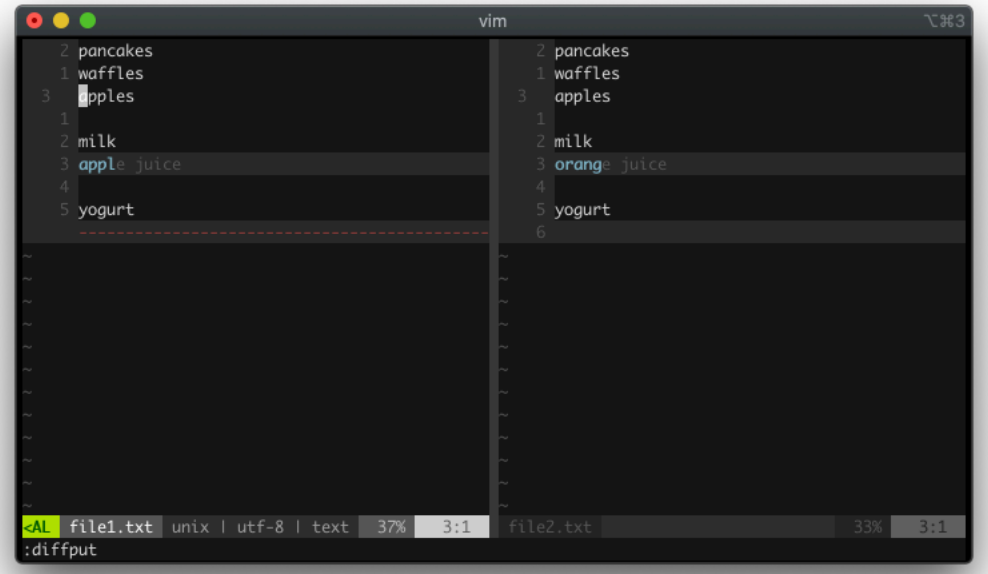
```
vim -d file1.txt file2.txt
```



Basic diffing with Vim

`vimdiff` displays two buffers side-by-side. On the left is `file1.txt` and on the right is `file2.txt`. The first differences (apples and oranges) are highlighted on both lines.

Suppose you want to make the second buffer to have apples, not oranges. To transfer the content from your current position (you're currently on `file1.txt`) to `file2.txt`, first go to the next diff with `]c` (to jump to the previous diff window, use `[c`). The cursor should be on apples now. Run `:diffput`. Both files should now have apples.



Diffing apples

If you need to transfer the text from the other buffer (orange juice, `file2.txt`) to replace the text on the current buffer (apple juice, `file1.txt`), with your cursor still on `file1.txt` window, first go to the next diff with `]c`. Your cursor now should be on apple juice. Run `:diffget` to get the orange juice from another buffer to replace apple juice in our buffer.

`:diffput` *puts out* the text from the current buffer to another buffer. `:diffget` *gets* the text from another buffer to the current buffer.

If you have multiple buffers, you can run `:diffput fileN.txt` and `:diffget fileN.txt` to target the the fileN buffer.

Vim As A Merge Tool

> "I love resolving merge conflicts!" - Nobody

I don't know anyone who likes resolving merge conflicts. However, they are inevitable. In this section, you will learn how to leverage Vim as a merge conflict resolution tool.

First, change the default merge tool to use `vimdiff` by running:

```
git config merge.tool vimdiff
git config merge.conflictstyle diff3
git config mergetool.prompt false
```

Alternatively, you can modify the `~/.gitconfig` directly (by default it should be in root, but yours might be in different place). The commands above should modify your gitconfig to look like the setting below, if you haven't run them already, you can also manually edit your gitconfig.

```
[core]
  editor = vim
[merge]
  tool = vimdiff
  conflictstyle = diff3
[difftool]
  prompt = false
```

Let's create a fake merge conflict to test this out. Create a directory `/food` and make it a git repository:

```
git init
```

Add a file, `breakfast.txt`. Inside:

```
pancakes
waffles
oranges
```

Add the file and commit it:

```
git add .
git commit -m "Initial breakfast commit"
```

Next, create a new branch and call it apples branch:

```
git checkout -b apples
```

Change the `breakfast.txt`:

```
pancakes
waffles
apples
```

Save the file, then add and commit the change:

```
git add .
git commit -m "Apples not oranges"
```

Great. Now you have oranges in the master branch and apples in the apples branch. Let's return to the master branch:

```
git checkout master
```

Inside `breakfast.txt`, you should see the base text, oranges. Let's change it to grapes because they are in season right now:

```
pancakes
waffles
grapes
```

Save, add, and commit:

```
git add .
git commit -m "Grapes not oranges"
```

Now you are ready to merge the apples branch into the master branch:

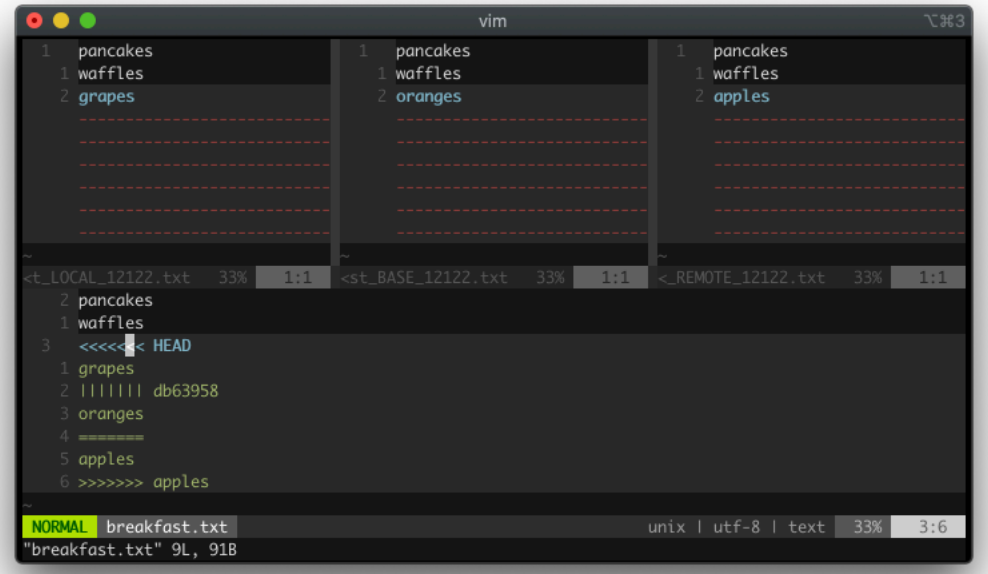
```
git merge apples
```

You should see an error:

```
Auto-merging breakfast.txt
CONFLICT (content): Merge conflict in breakfast.txt
Automatic merge failed; fix conflicts and then commit the result.
```

A conflict, great! Let's resolve the conflict using our newly-configured `mergetool`. Run:

```
git mergetool
```



Three-way mergetool with Vim

Vim displays four windows. Pay attention to the top three:

- `LOCAL` contains `grapes`. This is the change in "local", what you are merging into.
- `BASE` contains `oranges`. This is the common ancestor between `LOCAL` and `REMOTE` to compare how they diverge.
- `REMOTE` contains `apples`. This is what is being merged into.

At the bottom (the fourth window) you see:

```
pancakes
waffles
<<<<<<< HEAD
grapes
||||||| db63958
oranges
=======
apples
>>>>>>> apples
```

The fourth window contains the merge conflict texts. With this setup, it is easier to see what change each environment has. You can see the content from `LOCAL`, `BASE`, and `REMOTE` at the same time.

Your cursor should be on the fourth windows, on the highlighted area. To get the change from `LOCAL` (grapes), run `:diffget LOCAL`. To get the change from `BASE` (oranges), run `:diffget BASE` and to get the change from `REMOTE` (apples), run `:diffget REMOTE`.

In this case, let's get the change from `LOCAL`. Run `:diffget LOCAL`. The fourth window will now have grapes. Save and exit all files (`:wqall`) when you are done. That wasn't bad, right?

If you notice, you also have a file `breakfast.txt.orig` now. Git creates a backup file in case things don't go well. If you don't want git to create a backup during a merge, run:

```
git config --global mergetool.keepBackup false
```

Git Inside Vim

Vim does not have a native git feature built-in. One way to run git commands from Vim is to use the bang operator, `!`, in the command-line mode.

Any git command can be run with `!`:

```
:!git status
:!git commit
:!git diff
:!git push origin master
```

You can also use Vim's `%` (current buffer) or `#` (other buffer) conventions:

```
:!git add %         " git add current file
:!git checkout #    " git checkout the other file
```

One Vim trick you can use to add multiple files in different Vim window is to run:

```
:windo !git add %
```

Then make a commit:

```
:!git commit "Just git-added everything in my vim window, cool"
```

The `windo` command is one of Vim's "do" commands, similar to `argdo` that you saw previously. `windo` executes the command on each window.

Alternatively, you can also use `bufdo !git add %` to git add all buffers or `argdo !git add %` to git add all the file arguments, depending on your workflow.

Plugins

There are many Vim plugins for git support. Below is a list of some of the popular git-related plugins for Vim (there is probably more at the time you read this):

- vim-gitgutter[37]
- vim-signify[38]
- vim-fugitive[39]
- gv.vim[40]
- vimagit[41]
- vim-twiggy[42]
- rhubarb[43]

One of the most popular ones is vim-fugitive. For the remaining of the chapter, I will go over a several git workflow using this plugin.

Vim-fugitive

The vim-fugitive plugin allows you to run the git CLI without leaving the Vim editor. You will find that some commands are better when executed from inside Vim.

To get started, install the vim-fugitive with a vim plugin manager (vim-plug[44], vundle[45], dein.vim[46], etc).

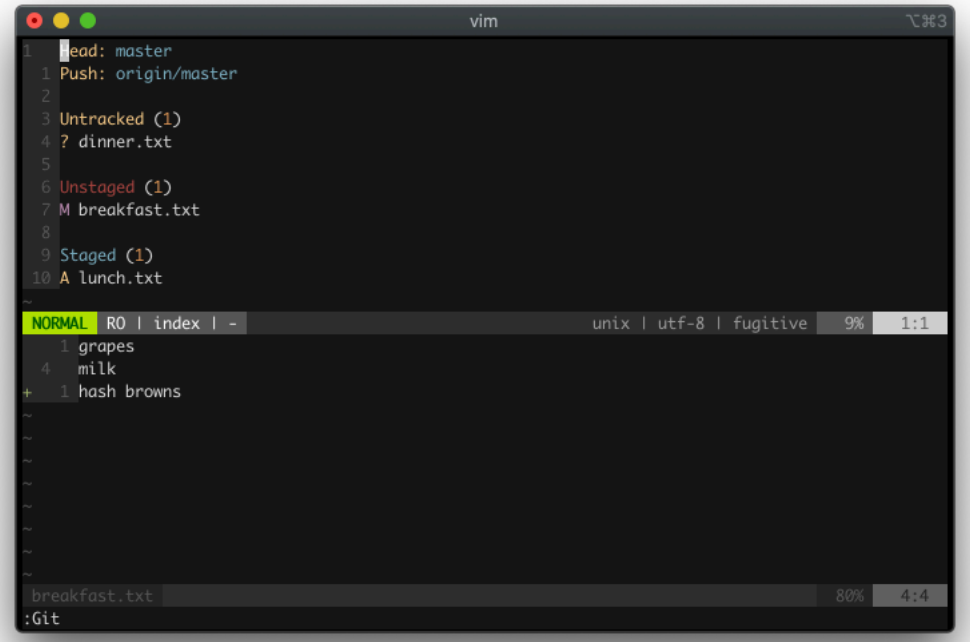
Git Status

When you run the `:Git` command without any parameters, vim-fugitive displays a git summary window. It shows the untracked, unstaged, and staged file(s). While in this "`git status`" mode, you can do several things:

- `Ctrl-N` / `Ctrl-P` to go up or down the file list.
- `-` to stage or unstage the file name under the cursor.
- `s` to stage the file name under the cursor.

[37] https://github.com/airblade/vim-gitgutter
[38] https://github.com/mhinz/vim-signify
[39] https://github.com/tpope/vim-fugitive
[40] https://github.com/junegunn/gv.vim
[41] https://github.com/jreybert/vimagit
[42] https://github.com/sodapopcan/vim-twiggy
[43] https://github.com/tpope/vim-rhubarb
[44] https://github.com/junegunn/vim-plug
[45] https://github.com/VundleVim/Vundle.vim
[46] https://github.com/Shougo/dein.vim

- `u` to unstage the file name under the cursor.
- `>` / `<` to display or hide an inline diff of the file name under the cursor.



Fugitive Git

For more, check out `:h fugitive-staging-maps`.

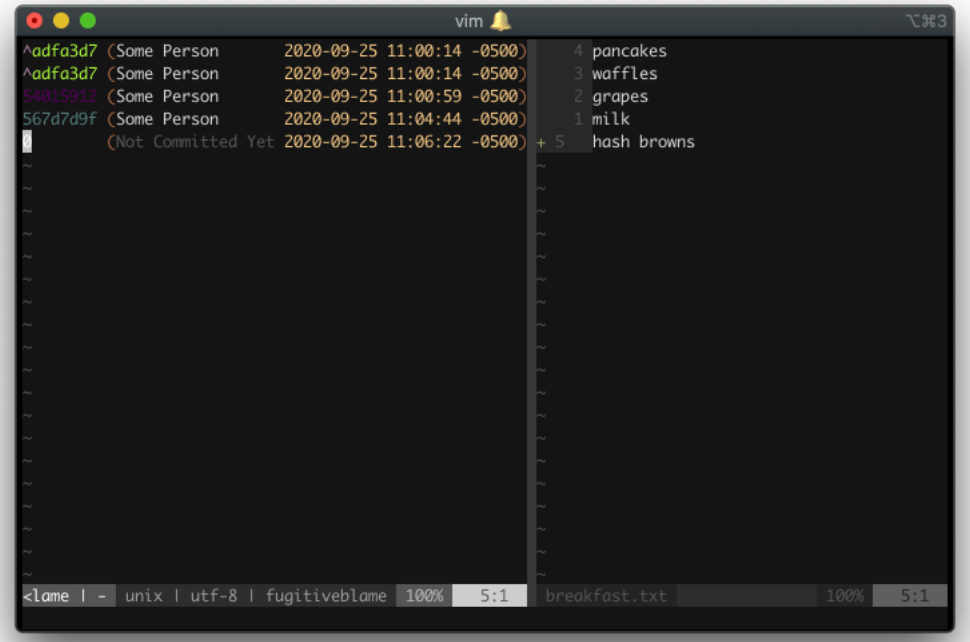
Git Blame

When you run the `:Git blame` command from the current file, vim-fugitive displays a split blame window. This can be useful to find the person responsible for writing that buggy line of code so you can yell at him / her (just kidding).

Some things you can do while in this `"git blame"` mode:

- `q` to close the blame window.
- `A` to resize the author column.
- `C` to resize the commit column.
- `D` to resize the date / time column.

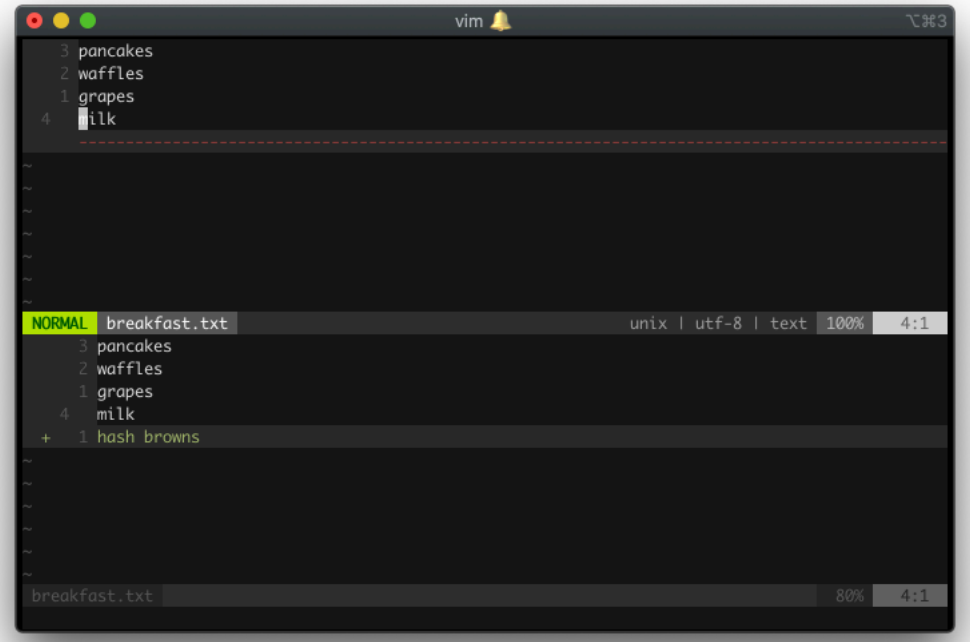
For more, check out `:h :Git_blame`.



Fugitive Git Blame

Gdiffsplit

When you run the `:Gdiffsplit` command, vim-fugitive runs a `vimdiff` of the current file's latest changes against the index or work tree. If you run `:Gdiffsplit <commit>`, vim-fugitive runs a `vimdiff` against that file inside `<commit>`.



Fugitive Gdiffsplit

Because you are in a `vimdiff` mode, you can *get* or *put* the diff with `:diffput` and `:diffget`.

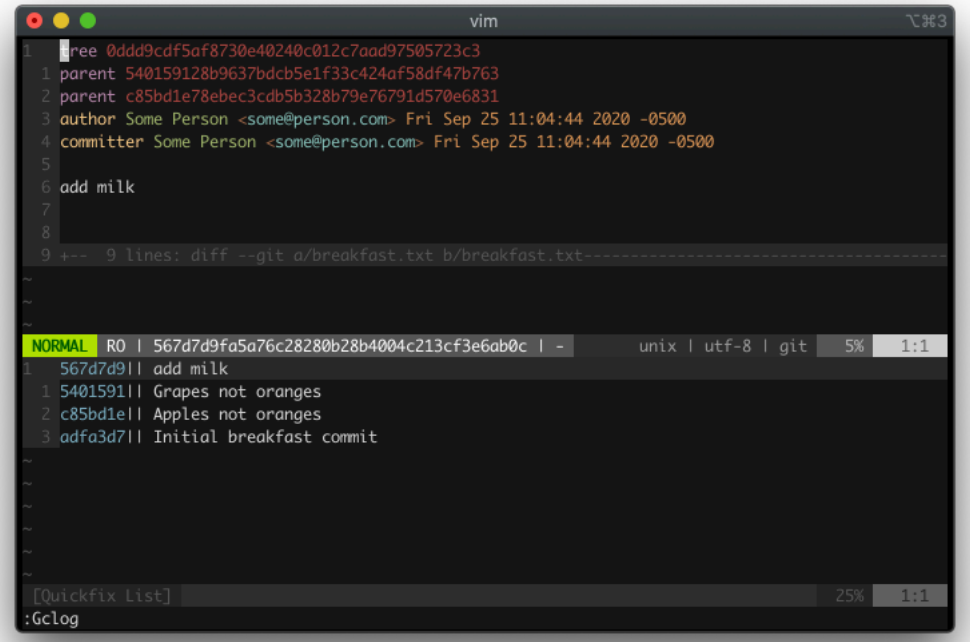
Gwrite And Gread

When you run the `:Gwrite` command in a file after you make changes, vim-fugitive stages the changes. It is like running `git add <current-file>`.

When you run the `:Gread` command in a file after you make changes, vim-fugitive restores the file to the state prior to the changes. It is like running `git checkout <current-file>`. One advantage of running `:Gread` is the action is undo-able. If, after you run `:Gread`, you change your mind and want to keep the old change, you can just run undo (`u`) and Vim will undo the `:Gread` action. This would not have been possible if you had run `git checkout <current-file>` from the CLI.

Gclog

When you run the `:Gclog` command, vim-fugitive displays the commit history. It is like running the `git log` command. Vim-fugitive uses Vim's quickfix to accomplish this, so you can use `:cnext` and `:cprevious` to traverse to the next or previous log information. You can open and close the log list with `:copen` and `:cclose`.



Fugitive Git Log

While in this `"git log"` mode, you can do two things:

- View the tree.
- Visit the parent (the previous commit).

You can pass to `:Gclog` arguments just like the `git log` command. If your project has a long commit history and you only need to view the last three commits, you can run `:Gclog -3`. If you need to filter it based on the committer's date, you can run something like `:Gclog --after="January 1" --before="March 14"`.

More Vim-Fugitive

These are only a few examples of what vim-fugitive can do. To learn more about vim-fugitive, check out `:h fugitive.txt`. Most of the popular git commands are probably optimized with vim-fugitive. You just have to look for them in the documentation.

If you are inside one of vim-fugitive's "special mode" (for example, inside `:Git` or `:Git blame` mode) and you want to learn what shortcuts are available, press `g?`. Vim-fugitive will display the appropriate `:help` window for the mode you are in. Neat!

Learn Vim And Git The Smart Way

You may find vim-fugitive to be a good compliment to your workflow (or not). Regardless, I would strongly encourage you to check out all the plugins listed above. There are probably others I didn't list. Go try them out.

One obvious way to get better with Vim-git integration is to read more about git. Git, on its own, is a vast topic and I am only showing a fraction of it. With that, let's *git going* (pardon the pun) and talk about how to use Vim to compile your code!

Ch19. Compile

Compiling is an important subject for many languages. In this chapter, you will learn how to compile from Vim. You will also look at ways to take advantage of Vim's `:make` command.

Compile From the Command Line

You can use the bang operator (`!`) to compile. If you need to compile your `.cpp` file with `g++`, run:

```
:!g++ hello.cpp -o hello
```

However, having to manually type the filename and the output filename each time is error-prone and tedious. A makefile is the way to go.

The Make Command

Vim has a `:make` command to run a makefile. When you run it, Vim looks for a makefile in the current directory to execute.

Create a file named `makefile` in the current directory and put these inside:

```
all:
        echo "Hello all"
foo:
        echo "Hello foo"
list_pls:
        ls
```

Run this from Vim:

```
:make
```

Vim executes it the same way as when you're running it from the terminal. The `:make` command accepts parameter just like the terminal make command. Run:

```
:make foo
" Outputs "Hello foo"

:make list_pls
" Outputs the ls command result
```

The `:make` command uses Vim's quickfix to store any error if you run a bad command. Let's run a nonexisting target:

```
:make dontexist
```

You should see an error running that command. To view that error, run the quickfix command `:copen` to view the quickfix window:

```
|| make: *** No rule to make target `dontexist'.  Stop.
```

Compiling With Make

Let's use the makefile to compile a basic `.cpp` program. First, let's create a `hello.cpp` file:

```
#include <iostream>

int main() {
    std::cout << "Hello!\\n";
    return 0;
}
```

Update your makefile to build and run a `.cpp` file:

```
all:
        echo "build, run"
build:
        g++ hello.cpp -o hello
run:
        ./hello
```

Now run:

```
:make build
```

The `g++` compiles `./hello.cpp` and creates `./hello`. Then run:

```
:make run
```

You should see `"Hello!"` printed on the terminal.

Different Make Program

When you run `:make`, Vim actually runs whatever command that is set under the `makeprg` option. If you run `:set makeprg?`, you'll see:

```
makeprg=make
```

The default `:make` command is the `make` external command. To change the `:make` command to execute `g++ {your-file-name}` each time you run it, run:

```
:set makeprg=g++\ %
```

The `\` is to escape the space after `g++`. The `%` symbol in Vim represents the current file. The command `g++\\ %` is equivalent to running `g++ hello.cpp`.

Go to `./hello.cpp` then run `:make`. Vim compiles `hello.cpp` and creates `a.out` because you didn't specify the output. Let's refactor it so it will name the compiled output with the name of the original file minus the extension. Run or add this to vimrc:

```
set makeprg=g++\ %\ -o\ %<
```

The breakdown:

- `g++\ %` is the same as above. It is equivalent to running `g++ <your-file>`.
- `-o` is the output option.
- `%<` in Vim represents the current file name without an extension (`hello.cpp` becomes `hello`).

When you run `:make` from inside `./hello.cpp`, it is compiled into `./hello`. To quickly execute `./hello` from inside `./hello.cpp`, run `:!./%<`. Again, this is the same as running `:!./{current-file-name-minus-the-extension}`.

For more, check out `:h :compiler` and `:h write-compiler-plugin`.

Auto-compile On Save

You can make life even easier by automating compilation. Recall that you can use Vim's `autocmd` to trigger automatic actions based on certain events. To automatically compile `.cpp` files on each save add this on your vimrc:

```
autocmd BufWritePost *.cpp make
```

Each time you save inside a `.cpp` file, Vim executes the `make` command.

Switching Compiler

Vim has a `:compiler` command to quickly switch compilers. Your Vim build probably comes with several pre-built compiler configurations. To check what compilers you have, run:

```
:e $VIMRUNTIME/compiler/<Tab>
```

You should see a list of compilers for different programming languages.

To use the `:compiler` command, suppose you have a ruby file, `hello.rb` and inside it has:

```
puts "Hello ruby"
```

Recall that if you run `:make`, Vim executes whatever command is assigned to `makeprg` (default is `make`). If you run:

```
:compiler ruby
```

Vim runs the `$VIMRUNTIME/compiler/ruby.vim` script and changes the `makeprg` to use the `ruby` command. Now if you run `:set makeprg?`, it should say `makeprg=ruby` (this depends on what is inside your `$VIMRUNTIME/compiler/ruby.vim` file or if you have another custom ruby compilers. Yours might be different). The `:compiler {your-lang}` command allows you to switch to different compilers quickly. This is useful if your project uses multiple languages.

You don't have to use the `:compiler` and `makeprg` to compile a program. You can run a test script, lint a file, send a signal, or anything you want.

Creating A Custom Compiler

Let's create a simple Typescript compiler. Install Typescript (`npm install -g typescript`) to your machine. You should now have the `tsc` command. If you haven't played with typescript before, `tsc` compiles a Typescript file into a Javascript file. Suppose that you have a file, `hello.ts`:

```
const hello = "hello";
console.log(hello);
```

If you run `tsc hello.ts`, it will compile into `hello.js`. However, if you have the following expressions inside `hello.ts`:

```
const hello = "hello";
hello = "hello again";
console.log(hello);
```

This will throw an error because you can't mutate a `const` variable. Running `tsc hello.ts` will throw an error:

```
hello.ts:2:1 - error TS2588: Cannot assign to 'person' because it is a constant.

2 person = "hello again";
  ~~~~~~


Found 1 error.
```

To create a simple Typescript compiler, in your `~/.vim/` directory, add a `compiler` directory (`~/.vim/compiler/`), then create a `typescript.vim` file (`~/.vim/compiler/typescript.vim`). Put this inside:

```
CompilerSet makeprg=tsc
CompilerSet errorformat=%f:\ %m
```

The first line sets the `makeprg` to run the `tsc` command. The second line sets the error format to display the file (`%f`), followed by a literal colon (`:`) and an escaped space (`\ `), followed by the error message (`%m`). To learn more about the error formatting, check out `:h errorformat`.

You should also read some of the pre-made compilers to see how others do it. Check out `:e $VIMRUNTIME/compiler/<some-language>.vim`.

Because some plugins may interfere with the Typescript file, let's open the `hello.ts` without any plugin, using the `--noplugin` flag:

```
vim --noplugin hello.ts
```

Check the `makeprg`:

```
:set makeprg?
```

It should say the default `make` program. To use the new Typescript compiler, run:

```
:compiler typescript
```

When you run `:set makeprg?`, it should say `tsc` now. Let's put it to the test. Run:

```
:make %
```

Recall that `%` means the current file. Watch your Typescript compiler work as expected! To see the list of error(s), run `:copen`.

Async Compiler

Sometimes compiling can take a long time. You don't want to be staring at a frozen Vim while waiting for your compilation process to finish. Wouldn't it be nice if you can compile asynchronously so you can still use Vim during compilation?

Luckily there are plugins to run async processes. The two big ones are:

- vim-dispatch[47]
- asyncrun.vim[48]

In the remaining of this chapter, I will go over vim-dispatch, but I would strongly encourage you to try all of them out there.

Vim and NeoVim actually supports async jobs, but they are beyond the scope of this chapter. If you're curious, check out `:h job-channel-overview.txt`.

Plugin: Vim-dispatch

Vim-dispatch has several commands, but the two main ones are `:Make` and `:Dispatch` commands.

Async Make

Vim-dispatch's `:Make` command is similar to Vim's `:make`, but it runs asynchronously. If you are in a Javascript project and you need to run `npm t`, you might attempt to set your makeprg to be:

```
:set makeprg=npm\\ t
```

If you run:

```
:make
```

Vim will execute `npm t`, but you will be staring at the frozen screen while your JavaScript test runs. With vim-dispatch, you can just run:

[47] https://github.com/tpope/vim-dispatch
[48] https://github.com/skywind3000/asyncrun.vim

```
:Make
```

Vim will run `npm t` asynchronously. This way, while `npm t` is running on a background process, you can continue doing whatever you were doing. Awesome!

Async Dispatch

The `:Dispatch` command is like the `:compiler` and the `:!` command. It can run any external command asynchronously in Vim.

Assume that you are inside a ruby spec file and you need to run a test. Run:

```
:Dispatch bundle exec rspec %
```

Vim will asynchronously run the `rspec` command against the current file (`%`).

Automating Dispatch

Vim-dispatch has `b:dispatch` buffer variable that you can configure to evaluate specific command automatically. You can leverage it with `autocmd`. If you add this in your vimrc:

```
autocmd BufEnter *_spec.rb let b:dispatch = 'bundle exec rspec %'
```

Now each time you enter a file (`BufEnter`) that ends with `_spec.rb`, running `:Dispatch` automatically executes `bundle exec rspec {your-current-ruby-spec-file}`.

Learn Compile The Smart Way

In this chapter, you learned that you can use the `make` and `compiler` commands to run *any* process from inside Vim asynchronously to complement your programming workflow. Vim's ability to extend itself with other programs makes it powerful.

Ch20. Views, Sessions, And Viminfo

After you worked on a project for a while, you may find the project to gradually take shape with its own settings, folds, buffers, layouts, etc. It's like decorating your apartment after living in it for a while. The problem is, when you close Vim, you lose those changes. Wouldn't it be nice if you can keep those changes so the next time you open Vim, it looks just like you had never left?

In this chapter, you will learn how use View, Session, and Viminfo to preserve a "snapshot" of your projects.

View

A View is the smallest subset of the three (View, Session, Viminfo). It is a collection of settings for one window. If you spend a long time working on a window and you want to preserve the maps and folds, you can use a View.

Let's create a file called `foo.txt`:

```
foo1
foo2
foo3
foo4
foo5
foo6
foo7
foo8
foo9
foo10
```

In this file, create three changes:

1. On line 1, create a manual fold `zf4j` (fold the next 4 lines).
2. Change the `number` setting: `setlocal nonumber norelativenumber`. This will remove the number indicators on the left side of the window.
3. Create a local mapping to go down two lines each time you press `j` instead of one: `:nnoremap <buffer> j jj`.

Your file should look like this:

```
+-- 5 lines: foo1 -----
foo6
foo7
foo8
foo9
foo10
```

Configuring View Attributes

Run:

```
:set viewoptions?
```

By default it should say (yours may look different depending on your vimrc):

```
viewoptions=folds,cursor,curdir
```

Let's configure `viewoptions`. The three attributes you want to preserve are the folds, the maps, and the local set options. If your setting looks like mine, you already have the `folds` option. You need to tell View to remember the `localoptions`. Run:

```
:set viewoptions+=localoptions
```

To learn what other options are available for `viewoptions`, check out `:h viewoptions`. Now if you run `:set viewoptions?`, you should see:

```
viewoptions=folds,cursor,curdir,localoptions
```

Saving The View

With the `foo.txt` window properly folded and having `nonumber norelativenumber` options, let's save the View. Run:

```
:mkview
```

Vim creates a View file.

View Files

You might wonder, "Where did Vim save this View file?" To see where Vim saves it, run:

```
:set viewdir?
```

The default should say `~/.vim/view` (if you have a different OS, it might show a different path. Check out `:h viewdir` for more). If you want to change it to a different path, add this into your vimrc:

```
set viewdir=$HOME/else/where
```

Loading The View File

Close the `foo.txt` if you haven't, then open `foo.txt` again. **You should see the original text without the changes**. That's expected.

To restore the state, you need to load the View file. Run:

```
:loadview
```

Now you should see:

```
+-- 5 lines: foo1 -----
foo6
foo7
foo8
foo9
foo10
```

The folds, local settings, and local mappings are restored. If you notice, your cursor should also be on the line where you left it when you ran `:mkview`. As long as you have the `cursor` option, View also remembers your cursor position.

Multiple Views

Vim lets you save 9 numbered Views (1-9).

Suppose you want to make an additional fold (say you want to fold the last two lines) with `:9,10 fold`. Let's save this as View 1. Run:

```
:mkview 1
```

If you want to make one more fold with `:6,7 fold` and save it as a different View, run:

```
:mkview 2
```

Close the file. When you open `foo.txt` and you want to load View 1, run:

```
:loadview 1
```

To load View 2, run:

```
:loadview 2
```

To load the original View, run:

```
:loadview
```

Automating View Creation

One of the worst things that can happen is, after spending countless hours organizing a large file with folds, you accidentally close the window and lose all fold information. To prevent this, you might want to automatically create a View each time you close a buffer. Add this in your vimrc:

```
autocmd BufWinLeave *.txt mkview
```

Additionally, it might be nice to load View when you open a buffer:

```
autocmd BufWinEnter *.txt silent loadview
```

Now you don't have to worry about creating and loading View anymore when you are working with `txt` files. Keep in mind that over time, your `~/.vim/view` might start to accumulate View files. It's good to clean it up once every few months.

Sessions

If a View saves the settings of a window, a Session saves the information of all windows (including the layout).

Creating A New Session

Suppose you are working with these 3 files in a `foobarbaz` project:

Inside `foo.txt`:

```
foo1
foo2
foo3
foo4
foo5
foo6
foo7
foo8
foo9
foo10
```

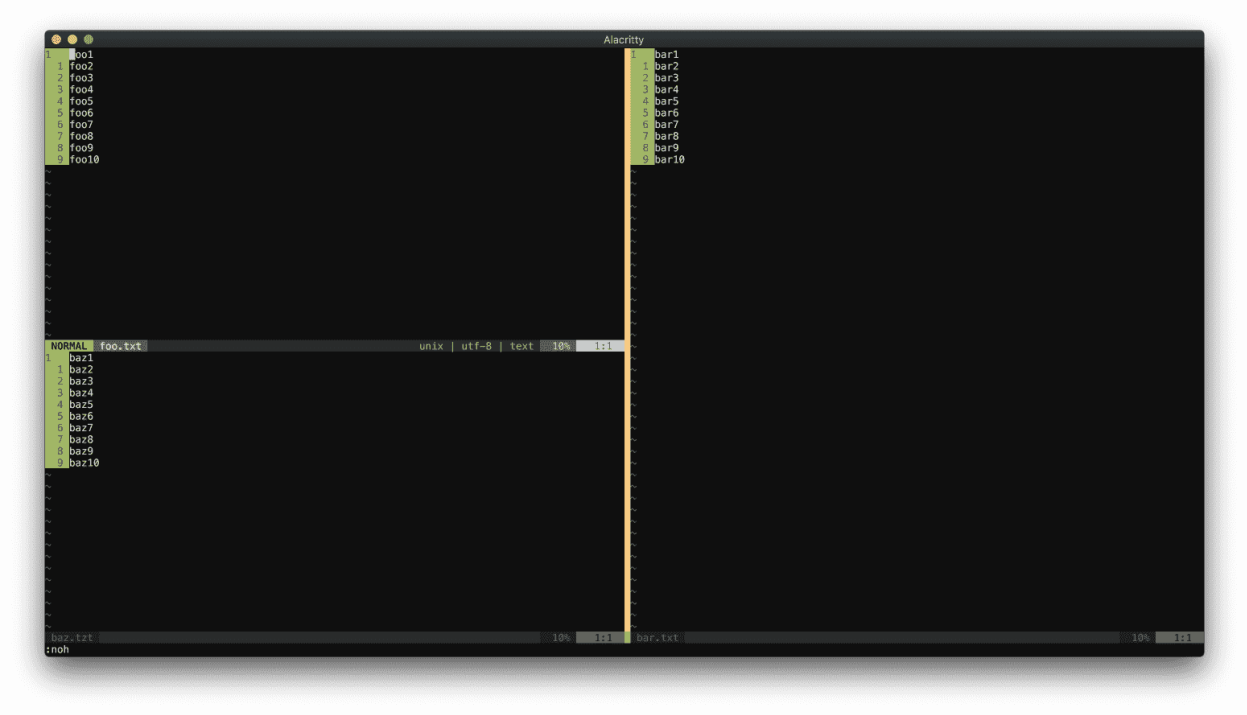
Inside `bar.txt`:

```
bar1
bar2
bar3
bar4
bar5
bar6
bar7
bar8
bar9
bar10
```

Inside `baz.txt`:

```
baz1
baz2
baz3
baz4
baz5
baz6
baz7
baz8
baz9
baz10
```

Let's say that your windows layout look like the following (using strategically placed `split` and `vsplit`):



Session Layout

To preserve this look, you need to save the Session. Run:

```
:mksession
```

Unlike `mkview` where it saves to `~/.vim/view` by default, `mksession` saves a Session file (`Session.vim`) in the current directory. Check out the file if you're curious what's inside.

If you want to save the Session file somewhere else, you can pass an argument to `mksession`:

```
:mksession ~/some/where/else.vim
```

If you want to overwrite the existing Session file, call the command with a `!` (`:mksession! ~/some/where/else.vim`).

Loading A Session

To load a Session, run:

```
:source Session.vim
```

Now Vim looks like just the way you left it! Alternatively, you can also load a Session file from the terminal:

```
vim -S Session.vim
```

Configuring Session Attributes

You can configure the attributes Session saves. To see what is currently being saved, run:

```
:set sessionoptions?
```

Mine says:

```
blank,buffers,curdir,folds,help,tabpages,winsize,terminal
```

If you don't want to save `terminal` when you save a Session, remove it from the session options. Run:

```
:set sessionoptions-=terminal
```

If you want to add an `options` when you save a Session, run:

```
:set sessionoptions+=options
```

Here are some attributes that `sessionoptions` can store:

- `blank` stores empty windows
- `buffers` stores buffers
- `folds` stores folds
- `globals` stores global variables (must start with an uppercase letter and contain at least one lowercase letter)
- `options` stores options and mappings
- `resize` stores window lines and columns
- `winpos` stores window position
- `winsize` stores window sizes
- `tabpages` stores tabs
- `unix` stores files in Unix format

For the complete list check out `:h 'sessionoptions'`.

Session is a useful tool to preserve your project's external attributes. However, some internal attributes aren't saved by Session, like local marks, registers, histories, etc. To save them, you need to use Viminfo!

Viminfo

If you notice, after yanking a word into register a and quitting Vim, the next time you open Vim you still that text stored in the register. This is actually a work of Viminfo. Without it, Vim won't remember the register after you close Vim.

If you use Vim 8 or higher, Vim enables Viminfo by default, so you may have been using Viminfo this whole time without knowing it!

You might ask: "What does Viminfo save? How does it differ from Session?"

To use Viminfo, first you need to have `+viminfo` feature available (`:version`). Viminfo stores:

- The command-line history.
- The search string history.
- The input-line history.
- Contents of non-empty registers.
- Marks for several files.
- File marks, pointing to locations in files.
- Last search / substitute pattern (for 'n' and '&').
- The buffer list.
- Global variables.

In general, Session stores the "external" attributes and Viminfo the "internal" attributes.

Unlike Session where you can have one Session file per project, you normally will use one Viminfo file per computer. Viminfo is project-agnostic.

The default Viminfo location for Unix is `$HOME/.viminfo` (`~/.viminfo`). If you use a different OS, your Viminfo location might be different. Check out `:h viminfo-file-name`. Each time you make "internal" changes, like yanking a text into a register, Vim automatically updates the Viminfo file.

Make sure that you have `nocompatible` option set (`set nocompatible`), otherwise your Viminfo will not work.

Writing And Reading Viminfo

Although you will use only one Viminfo file, you can create multiple Viminfo files. To write a Viminfo file, use the `:wviminfo` command (`:wv` for short).

```
:wv ~/.viminfo_extra
```

To overwrite an existing Viminfo file, add a bang to the `wv` command:

```
:wv! ~/.viminfo_extra
```

By default Vim will read from ~/.viminfo file. To read from a different Viminfo file, run :rviminfo, or :rv for short:

```
:rv ~/.viminfo_extra
```

To start Vim with a different Viminfo file from the terminal, use the i flag:

```
vim -i viminfo_extra
```

If you use Vim for different tasks, like coding and writing, you can create a Viminfo optimized for writing and another for coding.

```
vim -i viminfo_writing
```

```
vim -i viminfo_coding
```

Starting Vim Without Viminfo

To start Vim without Viminfo, you can run from the terminal:

```
vim -i NONE
```

To make it permanent, you can add this in your vimrc file:

```
set viminfo="NONE"
```

Configuring Viminfo Attributes

Similar to viewoptions and sessionoptions, you can instruct what attributes to save with the viminfo option. Run:

```
:set viminfo?
```

You will get:

```
!,'100,<50,s10,h
```

This looks cryptic. Let's break it down:

- `!` saves global variables that start with an uppercase letter and don't contain lowercase letters. Recall that `g:` indicates a global variable. For example, if at some point you wrote the assignment `let g:FOO = "foo"`, Viminfo will save the global variable `FOO`. However if you did `let g:Foo = "foo"`, Viminfo will not save this global variable because it contains lowercase letters. Without `!`, Vim won't safe those global variables.
- `'100` represents marks. In this case, Viminfo will save the local marks (a-z) of the last 100 files. Be aware that if you tell Viminfo to save too many files, Vim can start slowing down. 1000 is a good number to have.
- `<50` tells Viminfo how many maximum lines are saved for each registers (50 in this case). If I yank 100 lines of text into register a (`"ay99j`) and close Vim, the next time I open Vim and paste from register a (`"ap`), Vim will only paste 50 lines max. If you don't give maximum line number, *all* lines will be saved. If you give it 0, nothing will be saved.
- `s10` sets a size limit (in kb) for a register. In this case, any register greater than 10kb size will be excluded.
- `h` disables highlighting (from `hlsearch`) when Vim starts.

There are other options that you can pass. To learn more, check out `:h 'viminfo'`.

Using Views, Sessions, And Viminfo The Smart Way

Vim has View, Session, and Viminfo to take different level of your Vim environment snapshots. For micro projects, use Views. For larger projects, use Sessions. You should take your time to check out all the options that View, Session, and Viminfo offers.

Create your own View, Session, and Viminfo for your own editing style. If you ever need to use Vim outside of your computer, you can just load your settings and you will immediately feel at home!

Ch21. Vimrc

In the previous chapters, you learned how to use Vim text editor. This is great, but not the whole thing. To use Vim more effectively, you will need to learn how to configure it. The best place to start is your vimrc. TBC
In the previous chapters, you learned how to use Vim. In this chapter, you will learn how to orgnize and configure vimrc.

How Vim Finds Vimrc

The conventional wisdom for vimrc is to add a `.vimrc` dotfile in the root directory `~/.vimrc` (it might be different depending on your OS).

Behind the scene, Vim looks at multiple places for a vimrc file. Here are the places that Vim checks:

- `$VIMINIT`
- `$HOME/.vimrc`
- `$HOME/.vim/vimrc`
- `$EXINIT`
- `$HOME/.exrc`
- `$VIMRUNTIME/default.vim`

When you start Vim, it will check the above six locations in that order for a vimrc file. The first found vimrc file will be used and the rest is ignored.

First Vim will look for a `$VIMINIT`. If there is nothing there, Vim will check for `$HOME/.vimrc`. If there is nothing there, Vim will check for `$HOME/.vim/vimrc`. If Vim finds it, it will stop looking and use `$HOME/.vim/vimrc`.

The first location, `$VIMINIT`, is an environment variable. By default it is undefined. If you want to use `~/dotfiles/testvimrc` as your `$VIMINIT` value, you can create an environment variable containing the path of that vimrc. After you run `export VIMINIT='let $MYVIMRC="$HOME/dotfiles/testvimrc" | source $MYVIMRC'`, Vim will now use `~/dotfiles/testvimrc` as your vimrc file.

The second location, `$HOME/.vimrc`, is the conventional path for many Vim users. `$HOME` in many cases is your root directory (`~`). If you have a `~/.vimrc` file, Vim will use this as your vimrc file.

The third, `$HOME/.vim/vimrc`, is located inside the `~/.vim` directory. You might have the `~/.vim` directory already for your plugins, custom scripts, or View files. Note that there is no dot in vimrc file name (`$HOME/.vim/.vimrc` won't work, but `$HOME/.vim/vimrc` will).

The fourth, `$EXINIT` works similar to `$VIMINIT`.

The fifth, `$HOME/.exrc` works similar to `$HOME/.vimrc`.

The sixth, `$VIMRUNTIME/defaults.vim` is the default vimrc that comes with your Vim build. In my case, I have Vim 8.2 installed using Homebrew, so my path is (`/usr/local/share/vim/vim82`). If Vim does not find any of the previous six vimrc files, it will use this file.

For the remaining of this chapter, I am assuming that the vimrc uses the `~/.vimrc` path.

What To Put In My Vimrc?

A question I asked when I started was, "What should I put in my vimrc?"

The answer is, "anything you want". The temptation to copy-paste other people's vimrc is real, but you should resist it. If you insist to use someone else's vimrc, make sure that you know what it does, why and how s/he uses it, and most importantly, if it is relevant to you. Just because someone uses it doesn't mean you'll use it too.

Basic Vimrc Content

In the nutshell, a vimrc is a collection of:

- Plugins
- Settings
- Custom Functions
- Custom Commands
- Mappings

There are other things not mentioned above, but in general, this covers most use cases.

Plugins

In the previous chapters, I have mentioned different plugins, like fzf.vim[49], vim-mundo[50], and vim-fugitive[51].

Ten years ago, managing plugins was a nightmare. However, with the rise of modern plugin managers, installing plugins can now be done in seconds. I am currently using vim-plug[52] as my plugin manager, so I will use it in this section. The concept should be similar with other popular plugin managers. I would strongly recommend you to check out different ones, such as:

[49] https://github.com/junegunn/fzf.vim
[50] https://github.com/simnalamburt/vim-mundo
[51] https://github.com/tpope/vim-fugitive
[52] https://github.com/junegunn/vim-plug

- vundle.vim[53]
- vim-pathogen[54]
- dein.vim[55]

There are more plugin managers than the ones listed above, feel free to look around. To install vim-plug, if you have a Unix machine, run:

```
curl -fLo ~/.vim/autoload/plug.vim --create-dirs https://raw.githubusercontent.com/j\
unegunn/vim-plug/master/plug.vim
```

To add new plugins, drop your plugin names (`Plug 'github-username/repository-name'`) between the `call plug#begin()` and the `call plug#end()` lines. So if you want to install `emmet-vim` and `nerdtree`, put the following snippet down in your vimrc:

```
call plug#begin('~/.vim/plugged')
  Plug 'mattn/emmet-vim'
  Plug 'preservim/nerdtree'
call plug#end()
```

Save the changes, source it (`:source %`), and run `:PlugInstall` to install them.

In the future if you need to remove unused plugins, you just need to remove the plugin names from the `call` block, save and source, and run the `:PlugClean` command to remove it from your machine.

Vim 8 has its own built-in package managers. You can check out `:h packages` for more information. In the next chapter, I will show you how to use it.

Settings

It is common to see a lot of `set` options in any vimrc. If you run the set command from the command-line mode, it is not permanent. You will lose it when you close Vim. For example, instead of running `:set relativenumber number` from the Command-line mode each time you run Vim, you could just put these inside vimrc:

```
set relativenumber number
```

Some settings require you to pass it a value, like `set tabstop=2`. Check out the help page for each setting to learn what kind of values it accepts.

You can also use a `let` instead of `set` (make sure to prepend it with `&`). With `let`, you can use an expression as a value. For example, to set the `'dictionary'` option to a path only if the path exists:

[53] https://github.com/VundleVim/Vundle.vim
[54] https://github.com/tpope/vim-pathogen
[55] https://github.com/Shougo/dein.vim

```
let s:english_dict = "/usr/share/dict/words"

if filereadable(s:english_dict)
  let &dictionary=s:english_dict
endif
```

You will learn about Vimscript assignments and conditionals in later chapters.

For a list of all possible options in Vim, check out `:h E355`.

Custom Functions

Vimrc is a good place for custom functions. You will learn how to write your own Vimscript functions in a later chapter.

Custom Commands

You can create a custom Command-line command with `command`.

To create a basic command `GimmeDate` to display today's date:

```
:command! GimmeDate echo call("strftime", ["%F"])
```

When you run `:GimmeDate`, Vim will display a date like "2021-01-1".

To create a basic command with an input, you can use `<args>`. If you want to pass to `GimmeDate` a specific time/date format:

```
:command! GimmeDate echo call("strftime", [<args>])

:GimmeDate "%F"
" 2020-01-01

:GimmeDate "%H:%M"
" 11:30
```

If you want to restrict the number of arguments, you can pass it `-nargs` flag. Use `-nargs=0` to pass no argument, `-nargs=1` to pass one argument, `-nargs=+` to pass at least one argument, `-nargs=*` to pass any number of arguments, and `-nargs=?` to pass `0` or one arguments. If you want to pass nth argument, use `-nargs=n` (where `n` is any integer).

`<args>` has two variants: `<f-args>` and `<q-args>`. The former is used to pass arguments to Vimscript functions. The latter is used to automatically convert user input to strings.

Using `args`:

```
:command! -nargs=1 Hello echo "Hello " . <args>
:Hello "Iggy"
" returns 'Hello Iggy'

:Hello Iggy
" Undefined variable error
```

Using `q-args`:

```
:command! -nargs=1 Hello echo "Hello " . <q-args>
:Hello Iggy
" returns 'Hello Iggy'
```

Using `f-args`:

```
:function! PrintHello(person1, person2)
:  echo "Hello " . a:person1 . " and " . a:person2
:endfunction

:command! -nargs=* Hello call PrintHello(<f-args>)

:Hello Iggy1 Iggy2
" returns "Hello Iggy1 and Iggy2"
```

The functions above will make a lot more sense once you get to the Vimscript functions chapter.

To learn more about command and args, check out `:h command` and `:args`.

Maps

If you find yourself repeatedly performing the same complex task, it is a good indicator that you should create a mapping for that task.

For example, I have these two mappings in my vimrc:

```
nnoremap <silent> <C-f> :GFiles<CR>

nnoremap <Leader>tn :call ToggleNumber()<CR>
```

On the first one, I map `Ctrl-F` to fzf.vim[56] plugin's `:Gfiles` command (quickly search for Git files). On the second one, I map `<Leader>tn` to call a custom function `ToggleNumber` (toggles

[56] https://github.com/junegunn/fzf.vim

norelativenumber and relativenumber options). The Ctrl-F mapping overwrites Vim's native page scroll. Your mapping will overwrite Vim controls if they collide. Because I almost never used that feature, I decided that it is safe to overwrite it.

By the way, I personally like to use <Space> as the leader key instead of Vim's default. To change your leader key, add this in your vimrc:

```
let mapleader = "\<space>"
```

The nnoremap command used above can be broken down into three parts:

- map is the map command.
- n represents the normal mode.
- nore means non-recursive.

At minimum, you could have used nmap instead of nnoremap (nmap <silent> <C-f> :Gfiles<CR>). However, it is a good practice to use the non-recursive variant to avoid potential infinite loop.

Here's what could happen if you don't map non-recursively. Suppose you want to add a mapping to B to add a semi-colon at the end of the line, then go back one WORD (recall that B n Vim is a normal-mode navigation key to go backward one WORD).

```
nmap B A;<esc>B
```

When you press B... oh no! Vim adds ; uncontrollably (interrupt it with Ctrl-C). Why did that happen? Because in the mapping A;<esc>B, the B does not refer to Vim's native B function (go back one WORD), but it refers to the mapped function. What you have is actually this:

```
A;<esc>A;<esc>A;<esc>A;esc>...
```

To solve this problem, you need to add a non-recursive map:

```
nnoremap B A;<esc>B
```

Now try calling B again. This time it successfully adds a ; at the end of the line and go back one WORD. The B in this mapping represents Vim's original B functionality.

Vim has different maps for different modes. If you want to create a map for insert mode to exit insert mode when you press jk:

```
inoremap jk <esc>
```

The other map modes are: map (Normal, Visual, Select, and Operator-pending), vmap (Visual and Select), smap (Select), xmap (Visual), omap (Operator-pending), map! (Insert and Command-line), lmap (Insert, Command-line, Lang-arg), cmap (Command-line), and tmap (terminal-job). I won't cover them in detail. To learn more, check out :h map.txt.

Create a map that's most intuitive, consistent, and easy-to-remember.

Organizing Vimrc

Over time, your vimrc will grow large and become convoluted. There are two ways to keep your vimrc to look clean:

- Split your vimrc into several files.
- Fold your vimrc file.

Splitting Your Vimrc

You can split your vimrc to multiple files using Vim's `source` command. This command reads command-line commands from the given file argument.

Let's create a file inside the `~/.vim` directory and name it `/settings` (`~/.vim/settings`). The name itself is arbitrary and you can name it whatever you like.

You are going to split it into four components:

- Third-party plugins (`~/.vim/settings/plugins.vim`).
- General settings (`~/.vim/settings/configs.vim`).
- Custom functions (`~/.vim/settings/functions.vim`).
- Key mappings (`~/.vim/settings/mappings.vim`) .

Inside `~/.vimrc`:

```
source $HOME/.vim/settings/plugins.vim
source $HOME/.vim/settings/configs.vim
source $HOME/.vim/settings/functions.vim
source $HOME/.vim/settings/mappings.vim
```

Inside `~/.vim/settings/plugins.vim`:

```
call plug#begin('~/.vim/plugged')
  Plug 'mattn/emmet-vim'
  Plug 'preservim/nerdtree'
call plug#end()
```

Inside `~/.vim/settings/configs.vim`:

```
set nocompatible
set relativenumber
set number
```

Inside `~/.vim/settings/functions.vim`:

```
function! ToggleNumber()
  if(&relativenumber == 1)
    set norelativenumber
  else
    set relativenumber
  endif
endfunc
```

Inside `~/.vim/settings/mappings.vim`:

```
inoremap jk <esc>
nnoremap <silent> <C-f> :GFiles<CR>
nnoremap <Leader>tn :call ToggleNumber()<CR>
```

Your vimrc should works as usual, but now it is only four lines long!

With this setup, you easily know where to go. If you need to add more mappings, add them to the `/mappings.vim` file. In the future, you can always add more directories as your vimrc grows. For example, if you need to create a setting for your colorschemes, you can add a `~/.vim/settings/themes.vim`.

Keeping One Vimrc File

If you prefer to keep one vimrc file to keep it portable, you can use the marker folds to keep it organized. Add this at the top of your vimrc:

```
" setup folds {{{
augroup filetype_vim
  autocmd!
  autocmd FileType vim setlocal foldmethod=marker
augroup END
" }}}
```

Vim can detect what kind of filetype the current buffer has (`:set filetype?`). If it is a `vim` filetype, you can use a marker fold method. Recall that a marker fold uses `{{{` and `}}}` to indicate the starting and ending folds.

Add `{{{` and `}}}` folds to the rest of your vimrc (don't forget to comment them with `"`):

```
" setup folds {{{
augroup filetype_vim
  autocmd!
  autocmd FileType vim setlocal foldmethod=marker
augroup END
" }}}

" plugins {{{
call plug#begin('~/.vim/plugged')
  Plug 'mattn/emmet-vim'
  Plug 'preservim/nerdtree'
call plug#end()
" }}}

" configs {{{
set nocompatible
set relativenumber
set number
" }}}

" functions {{{
function! ToggleNumber()
  if(&relativenumber == 1)
    set norelativenumber
  else
    set relativenumber
  endif
endfunc
" }}}

" mappings {{{
inoremap jk <esc>
nnoremap <silent> <C-f> :GFiles<CR>
nnoremap <Leader>tn :call ToggleNumber()<CR>
" }}}
```

Your vimrc should look like this:

```
+-- 6 lines: setup folds -----

+-- 6 lines: plugins ---------

+-- 5 lines: configs ---------

+-- 9 ilnes: functions -------

+-- 5 lines: mappings --------
```

Running Vim With Or Without Vimrc And Plugins

If you need to run Vim without both vimrc and plugins, run:

```
vim -u NONE
```

If you need to launch Vim without vimrc but with plugins, run:

```
vim -u NORC
```

If you need to run Vim with vimrc but without plugins, run:

```
vim --noplugin
```

If you need to run Vim with a *different* vimrc, say `~/.vimrc-backup`, run:

```
vim -u ~/.vimrc-backup
```

Configure Vimrc The Smart Way

Vimrc is an important component of Vim customization. A good way to start building your vimrc is by reading other people's vimrcs and gradually build it over time. The best vimrc is not the one that developer X uses, but the one that is tailored exactly to fit your thinking framework and editing style.

Ch22. Vim Packages

In the previous chapter, I mentioned using an external plugin manager to install plugins. Since version 8, Vim comes with its own built-in plugin manager called *packages*. In this chapter, you will learn how to use Vim packages to install plugins.

To see if your Vim build has the ability to use packages, run `:version` and look for `+packages` attribute. Alternatively, you can also run `:echo has('packages')` (if it returns `1`, then it has the packages ability).

Pack Directory

Check if you have a `~/.vim/` directory in the root path. If you don't, create one. Inside it, create a directory called `pack` (`~/.vim/pack/`). Vim automatically knows to search inside this directory for packages.

Two Types Of Loading

Vim package has two loading mechanisms: automatic and manual loading.

Automatic Loading

To load plugins automatically when Vim starts, you need to put them in the `start/` directory. The path looks like this:

```
~/.vim/pack/*/start/
```

Now you may ask, "What is the `*` between `pack/` and `start/`?" `*` is an arbitrary name and can be anything you want. let's name it `packdemo/`:

```
~/.vim/pack/packdemo/start/
```

Keep in mind that if you skip it and do something like this instead:

```
~/.vim/pack/start/
```

The package system won't work. It is imperative to put a name between `pack/` and `start/`.

For this demo, let's try to install the NERDTree[57] plugin. Go all the way to the `start/` directory (`cd ~/.vim/pack/packdemo/start/`) and clone the NERDTree repository:

[57] https://github.com/preservim/nerdtree

```
git clone https://github.com/preservim/nerdtree.git
```

That's it! You are all set. The next time you start Vim, you can immediately execute NERDTree commands like `:NERDTreeToggle`.

You can clone as many plugin repositories as you want inside the `~/.vim/pack/*/start/` path. Vim will automatically load each one. If you remove the cloned repository (`rm -rf nerdtree/`), that plugin will not be available anymore.

Manual Loading

To load plugins manually when Vim starts, you need to put them in the `opt/` directory. Similar to automatic loading, the path looks like this:

```
~/.vim/pack/*/opt/
```

Let's use the same `packdemo/` directory from earlier:

```
~/.vim/pack/packdemo/opt/
```

This time, let's install the killersheep[58] game (this requires Vim 8.2). Go to the `opt/` directory (`cd ~/.vim/pack/packdemo/opt/`) and clone the repository:

```
git clone https://github.com/vim/killersheep.git
```

Start Vim. The command to execute the game is `:KillKillKill`. Try running it. Vim will complain that it is not a valid editor command. You need to *manually* load the plugin first. Let's do that:

```
:packadd killersheep
```

Now try running the command again `:KillKillKill`. The command should work now.

You may wonder, "Why would I ever want to manually load packages? Isn't it better to automatically load everything at the start?"

Great question. Sometimes there are plugins that you won't use all the time, like that KillerSheep game. You probably don't need to load 10 different games and slow down Vim startup time. However, once in a while, when you are bored, you might want to play a few games. Use manual loading for nonessential plugins.

You can also use this to conditionally add plugins. Maybe you use both Neovim and Vim and there are plugins optimized for Neovim. You can add something like this in your vimrc:

[58] https://github.com/vim/killersheep

```
if has('nvim')
  packadd! neovim-only-plugin
else
  packadd! generic-vim-plugin
endif
```

Organizing packages

Recall that the requirement to use Vim's package system is to have either:

```
~/.vim/pack/*/start/
```

Or:

```
~/.vim/pack/*/opt/
```

The fact that * can be *any* name can be used to organize your packages. Suppose you want to group your plugins based on categories (colors, syntax, and games):

```
~/.vim/pack/colors/
~/.vim/pack/syntax/
~/.vim/pack/games/
```

You can still use `start/` and `opt/` inside each of the directories.

```
~/.vim/pack/colors/start/
~/.vim/pack/colors/opt/

~/.vim/pack/syntax/start/
~/.vim/pack/syntax/opt/

~/.vim/pack/games/start/
~/.vim/pack/games/opt/
```

Adding Packages The Smart Way

You may wonder if Vim package will make popular plugin managers like vim-pathogen, vundle.vim, dein.vim, and vim-plug obsolete.

The answer is, as always, "it depends."

I still use vim-plug because it makes it easy to add, remove or update plugins. If you use many plugins, it may be more convenient to use plugin managers because it is easy to update many simultaneously. Some plugin managers also offer asynchronous functionalities.

If you are a minimalist, try out Vim packages. If you a heavy plugin user, you may want to consider using a plugin manager.

Ch23. Vim Runtime

In the previous chapters, I mentioned that Vim automatically looks for special paths like `pack/` (Ch. 22) and `compiler/` (Ch. 19) inside the `~/.vim/` directory. These are examples of Vim runtime paths.

Vim has more runtime paths than these two. In this chapter, you will learn a high-level overview of these runtime paths. The goal of this chapter is to show you when they are called. Knowing this will allow you to understand and customize Vim further.

Runtime Path

In a Unix machine, one of your Vim runtime paths is `$HOME/.vim/` (if you have a different OS like Windows, your path might be different). To see what the runtime paths for different OS are, check out `:h 'runtimepath'`. In this chapter, I will use `~/.vim/` as the default runtime path.

Plugin Scripts

Vim has a plugin runtime path that executes any scripts in this directory once each time Vim starts. Do not confuse the name "plugin" with Vim external plugins (like NERDTree, fzf.vim, etc).

Go to `~/.vim/` directory and create a `plugin/` directory. Create two files: `donut.vim` and `chocolate.vim`.

Inside `~/.vim/plugin/donut.vim`:

```
echo "donut!"
```

Inside `~/.vim/plugin/chocolate.vim`:

```
echo "chocolate!"
```

Now close Vim. The next time you start Vim, you will see both `"donut!"` and `"chocolate!"` echoed. The plugin runtime path can be used for initializations scripts.

Filetype Detection

Before you start, to ensure that these detections work, make sure that your vimrc contains at least the following line:

```
filetype plugin indent on
```

Check out `:h filetype-overview` for more context. Essentially this turns on Vim's filetype detection.

When you open a new file, Vim usually knows what kind of file it is. If you have a file `hello.rb`, running `:set filetype?` returns the correct response `filetype=ruby`.

Vim knows how to detect "common" file types (Ruby, Python, Javascript, etc). But what if you have a custom file? You need to teach Vim to detect it and assign it with the correct file type.

There are two methods of detection: using file name and file content.

File Name Detection

File name detection detects a file type using the name of that file. When you open the `hello.rb` file, Vim knows it is a Ruby file from the `.rb` extension.

There are two ways you can do file name detection: using `ftdetect/` runtime directory and using `filetype.vim` runtime file. Let's explore both.

ftdetect/

Let's create an obscure (yet tasty) file, `hello.chocodonut`. When you open it and you run `:set filetype?`, since it is not a common file name extension Vim doesn't know what to make of it. It returns `filetype=`.

You need to instruct Vim to set all files ending with `.chocodonut` as a "chocodonut" file type. Create a directory named `ftdetect/` in the runtime root (`~/.vim/`). Inside, create a file and name it `chocodonut.vim` (`~/.vim/ftdetect/chocodonut.vim`). Inside this file, add:

```
autocmd BufNewFile,BufRead *.chocodonut set filetype=chocodonut
```

`BufNewFile` and `BufRead` are triggered whenever you create a new buffer and open a new buffer. `*.chocodonut` means that this event will only be triggered if the opened buffer has a `.chocodonut` filename extension. Finally, `set filetype=chocodonut` command sets the file type to be a chocodonut type.

Restart Vim. Now open `hello.chocodonut` file and run `:set filetype?`. It returns `filetype=chocodonut`.

Scrumptious! You can put as many files as you want inside `ftdetect/`. In the future, you can maybe add `ftdetect/strawberrydonut.vim`, `ftdetect/plaindonut.vim`, etc., if you ever decide to expand your donut file types.

There are actually two ways to set a file type in Vim. One is what you just used `set filetype=chocodonut`. The other way is to run `setfiletype chocodonut`. The former command `set filetype=chocodonut` will *always* set the file type to chocodonut type, while the latter command `setfiletype chocodonut` will only set the file type if no file type was set yet.

Filetype File

The second file detection method requires you to create a `filetype.vim` in the root directory (`~/.vim/filetype.vim`). Add this inside:

```
autocmd BufNewFile,BufRead *.plaindonut set filetype=plaindonut
```

Create a `hello.plaindonut` file. When you open it and run `:set filetype?`, Vim displays the correct custom file type `filetype=plaindonut`.

Holy pastry, it works! By the way, if you play around with `filetype.vim`, you may notice that this file is being run multiple times when you open `hello.plaindonut`. To prevent this, you can add a guard so the main script is run only once. Update the `filetype.vim`:

```
if exists("did_load_filetypes")
  finish
endif

augroup donutfiletypedetection
  autocmd! BufRead,BufNewFile *.plaindonut setfiletype plaindonut
augroup END
```

`finish` is a Vim command to stop running the rest of the script. The `"did_load_filetypes"` expression is *not* a built-in Vim function. It is actually a global variable from inside `$VIMRUNTIME/filetype.vim`. If you're curious, run `:e $VIMRUNTIME/filetype.vim`. You will find these lines inside:

```
if exists("did_load_filetypes")
  finish
endif

let did_load_filetypes = 1
```

When Vim calls this file, it defines `did_load_filetypes` variable and sets is to `1`. `1` is truthy in Vim. You should read the rest of the `filetype.vim` too. See if you can understand what it does when Vim calls it.

File Type Script

Let's learn how to detect and assign a file type based on the file content.

Suppose you have a collection of files without an agreeable extension. The only thing these files have in common is that they all start with the word "donutify" on the first line. You want to assign these files to a `donut` file type. Create new files named `sugardonut`, `glazeddonut`, and `frieddonut` (without extension). Inside each file, add this line:

```
donutify
```

When you run the `:set filetype?` from inside `sugardonut`, Vim doesn't know what file type to assign this file with. It returns `filetype=`.

In the runtime root path, add a `scripts.vim` file (`~/.vim/scripts.vim`). Inside it, add these:

```
if did_filetype()
  finish
endif

if getline(1) =~ '^\\<donutify\\>'
  setfiletype donut
endif
```

The function `getline(1)` returns the text on the first line. It checks if the first line starts with the word "donutify". The function `did_filetype()` is a Vim built-in function. It will return true when a file type related event is triggered at least once. It is used as a guard to stop re-running file type event.

Open the `sugardonut` file and run `:set filetype?`, Vim now returns `filetype=donut`. If you open another donut files (`glazeddonut` and `frieddonut`), Vim also identifies their file types as `donut` types.

Note that `scripts.vim` is only run when Vim opens a file with an unknown file type. If Vim opens a file with a known file type, `scripts.vim` won't run.

File Type Plugin

What if you want Vim to run chocodonut-specific scripts when you open a chocodonut file and to not run those scripts when opening plaindonut file?

You can do this with file type plugin runtime path (`~/.vim/ftplugin/`). Vim looks inside this directory for a file with the same name as the file type you just opened. Create a `chocodonut.vim` (`~/.vim/ftplugin/chocodonut.vim`):

```
echo "Calling from chocodonut ftplugin"
```

Create another ftplugin file, `plaindonut.vim` (`~/.vim/ftplugin/plaindonut.vim`):

```
echo "Calling from plaindonut ftplugin"
```

Now each time you open a chocodonut file type, Vim runs the scripts from `~/.vim/ftplugin/chocodonut.vim`. Each time you open a plaindonut file type, Vim runs the scripts from `~/.vim/ftplugin/plaindonut.vim`.

One warning: these files are run each time a buffer file type is set (`set filetype=chocodonut` for example). If you open 3 different chocodonut files, the scripts will be run a *total* of three times.

Indent Files

Vim has an indent runtime path that works similar to ftplugin, where Vim looks for a file named the same as the opened file type. The purpose of these indent runtime paths is to store indent-related codes. If you the file `~/.vim/indent/chocodonut.vim`, it will be executed only when you open a chocodonut file type. You can store indent-related codes for chocodonut files here.

Colors

Vim has a colors runtime path (`~/.vim/colors/`) to store color schemes. Any file that goes inside the directory will be displayed in the `:color` command-line command.

If you have a `~/.vim/colors/beautifulprettycolors.vim` file, when you run `:color` and press tab, you will see `beautifulprettycolors` as one of the color options. If you prefer to add your own color scheme, this is the place to go.

If you want to check out the color schemes other people made, a good place to visit is vimcolors[59].

Syntax Highlighting

Vim has a syntax runtime path (`~/.vim/syntax/`) to define syntax highlighting.

Suppose you have a `hello.chocodonut` file, inside it you have the following expressions:

```
(donut "tasty")
(donut "savory")
```

Although Vim now knows the correct file type, all texts have the same color. Let's add a syntax highlighting rule to highlight the "donut" keyword. Create a new chocodonut syntax file, `~/.vim/syntax/chocodonut.vim`. Inside it add:

```
syntax keyword donutKeyword donut

highlight link donutKeyword Keyword
```

Now reopen `hello.chocodonut` file. The `donut` keywords are now highlighted.

This chapter won't go over syntax highlighting in depth. It is a vast topic. If you are curious, check out `:h syntax.txt`.

The vim-polyglot[60] plugin is a great plugin that provides highlights for many popular programming languages.

[59] https://vimcolors.com/
[60] https://github.com/sheerun/vim-polyglot

Documentation

If you create a plugin, you will have to create your own documentation. You use the doc runtime path for that.

Let's create a basic documentation for chocodonut and plaindonut keywords. Create a `donut.txt` (`~/.vim/doc/donut.txt`). Inside, add these texts:

```
*chocodonut* Delicious chocolate donut

*plaindonut* No choco goodness but still delicious nonetheless
```

If you try to search for `chocodonut` and `plaindonut` (`:h chocodonut` and `:h plaindonut`), you won't find anything.

First, you need to run `:helptags` to generate new help entries. Run `:helptags ~/.vim/doc/`

Now if you run `:h chocodonut` and `:h plaindonut`, you will find these new help entries. Notice that the file is now read-only and has a "help" file type.

Lazy Loading Scripts

All of the runtime paths that you learned in this chapter were run automatically. If you want to manually load a script, use the autoload runtime path.

Create an autoload directory (`~/.vim/autoload/`). Inside that directory, create a new file and name it `tasty.vim` (`~/.vim/autoload/tasty.vim`). Inside it:

```
echo "tasty.vim global"

function tasty#donut()
  echo "tasty#donut"
endfunction
```

Note that the function name is `tasty#donut`, not `donut()`. The pound sign (`#`) is required when using the autoload feature. The function naming convention for the autoload feature is:

```
function fileName#functionName()
  ...
endfunction
```

In this case, the file name is `tasty.vim` and the function name is (technically) `donut`.

To invoke a function, you need the `call` command. Let's call that function with `:call tasty#donut()`.

The first time you call the function, you should see *both* echo messages ("tasty.vim global" and "tasty#donut"). The subsequent calls to `tasty#donut` function will only display "testy#donut" echo.

When you open a file in Vim, unlike the previous runtime paths, autoload scripts aren't loaded automatically. Only when you explicitly call `tasty#donut()`, Vim looks for the `tasty.vim` file and loads everything inside it, including the `tasty#donut()` function. Autoload is the perfect mechanism for functions that use extensive resources but you don't use often.

You can add as many nested directories with autoload as you want. If you have the runtime path `~/.vim/autoload/one/two/three/tasty.vim`, you can call the function with `:call one#two#three#tasty#donut()`.

After Scripts

Vim has an after runtime path (`~/.vim/after/`) that mirrors the structure of `~/.vim/`. Anything in this path is executed last, so developers usually use these paths for script overrides.

For example, if you want to overwrite the scripts from `plugin/chocolate.vim`, you can create `~/.vim/after/plugin/chocolate.vim` to put the override scripts. Vim will run the `~/.vim/after/plugin/chocolate.vim` *after* `~/.vim/plugin/chocolate.vim`.

$VIMRUNTIME

Vim has an environment variable `$VIMRUNTIME` for default scripts and support files. You can check it out by running `:e $VIMRUNTIME`.

The structure should look familiar. It contains many runtime paths you learned in this chapter.

Recall in Chapter 21, you learned that when you open Vim, it looks for a vimrc files in seven different locations. I said that the last location Vim checks is `$VIMRUNTIME/default.vim`. If Vim fails to find any uservimrc files, Vim uses a `default.vim` as vimrc.

Have you ever tried running Vim without syntax plugin like vim-polyglot and yet your file is still syntatically highlighted? That is because when Vim fails to find a syntax file from the runtime path, Vim looks for a syntax file from `$VIMRUNTIME` syntax directory.

To learn more, check out `:h $VIMRUNTIME`.

Runtimepath Option

To check your runtimepath, run `:set runtimepath?`

If you use Vim-Plug or popular external plugin managers, it should display a list of directories. For example, mine shows:

```
runtimepath=~/.vim,~/.vim/plugged/vim-signify,~/.vim/plugged/base16-vim,~/.vim/plugg\
ed/fzf.vim,~/.vim/plugged/fzf,~/.vim/plugged/vim-gutentags,~/.vim/plugged/tcomment_v\
im,~/.vim/plugged/emmet-vim,~/.vim/plugged/vim-fugitive,~/.vim/plugged/vim-sensible,\
~/.vim/plugged/lightline.vim, ...
```

One of the things plugin managers does is adding each plugin into the runtime path. Each runtime path can have its own directory structure similar to `~/.vim/`.

If you have a directory `~/box/of/donuts/` and you want to add that directory to your runtime path, you can add this to your vimrc:

```
set rtp+=$HOME/box/of/donuts/
```

If inside `~/box/of/donuts/`, you have a plugin directory (`~/box/of/donuts/plugin/hello.vim`) and a ftplugin (`~/box/of/donuts/ftplugin/chocodonut.vim`), Vim will run all scripts from `plugin/hello.vim` when you open Vim. Vim will also run `ftplugin/chocodonut.vim` when you open a chocodonut file.

Try this yourself: create an arbitrary path and add it to your runtimepath. Add some of the runtime paths you learned from this chapter. Make sure they work as expected.

Learn Runtime The Smart Way

Take your time reading it and play around with these runtime paths. To see how runtime paths are being used in the wild, go to the repository of one of your favorite Vim plugins and study its directory structure. You should be able to understand most of them now. Try to follow along and discern the big picture. Now that you understand Vim directory structure, you're ready to learn Vimscript.

Ch24. Vimscript Basic Data Types

In the next few chapters, you will learn about Vimscript, Vim's built-in programming language.

When learning a new language, there are three basic elements to look for:

- Primitives
- Means of Combination
- Means of Abstraction

In this chapter, you will learn Vim's primitive data types.

Data Types

Vim has 10 different data types:

- Number
- Float
- String
- List
- Dictionary
- Special
- Funcref
- Job
- Channel
- Blob

I will cover the first six data types here. In Ch. 27, you will learn about Funcref. For more about Vim data types, check out `:h variables`.

Following Along With Ex Mode

Vim technically does not have a built-in REPL, but it has a mode, Ex mode, that can be used like one. You can go to the Ex mode with `Q` or `gQ`. The Ex mode is like an extended command-line mode (it's like typing command-line mode commands non-stop). To quit the Ex mode, type `:visual`.

You can use either `:echo` or `:echom` on this chapter and the subsequent Vimscript chapters to code along. They are like `console.log` in JS or `print` in Python. The `:echo` command prints the evaluated expression you give. The `:echom` command does the same, but in addition, it stores the result in the message history.

```
:echom "hello echo message"
```

You can view the message history with:

```
:messages
```

To clear your message history, run:

```
:messages clear
```

Number

Vim has 4 different number types: decimal, hexadecimal, binary, and octal. By the way, when I say number data type, often this means an integer data type. In this guide, I will use the terms number and integer interchangeably.

Decimal

You should be familiar with the decimal system. Vim accepts positive and negative decimals. 1, -1, 10, etc. In Vimscript programming, you will probably be using the decimal type most of the time.

Hexadecimal

Hexadecimals start with `0x` or `0X`. Mnemonic: He**x**adecimal.

Binary

Binaries start with `0b` or `0B`. Mnemonic: **B**inary.

Octal

Octals start with `0`, `0o`, and `0O`. Mnemonic: **O**ctal.

Printing Numbers

If you `echo` either a hexadecimal, a binary, or an octal number, Vim automatically converts them to decimals.

```
:echo 42
" returns 42

:echo 052
" returns 42

:echo 0b101010
" returns 42

:echo 0x2A
" returns 42
```

Truthy and Falsy

In Vim, a `0` value is falsy and all non-`0` values are truthy.

The following will not echo anything.

```
:if 0
:  echo "Nope"
:endif
```

However, this will:

```
:if 1
:  echo "Yes"
:endif
```

Any values other than `0` is truthy, including negative numbers. `100` is truthy. `-1` is truthy.

Number Arithmetic

Numbers can be used to run arithmetic expressions:

```
:echo 3 + 1
" returns 4

: echo 5 - 3
" returns 2

:echo 2 * 2
" returns 4

:echo 4 / 2
" returns 2
```

When dividing a number with a remainder, Vim drops the remainder.

```
:echo 5 / 2
" returns 2 instead of 2.5
```

To get a more accurate result, you need to use a float number.

Float

Floats are numbers with trailing decimals. There are two ways to represent floating numbers: dot point notation (like 31.4) and exponent (3.14e01). Similar to numbers, you can use positive and negative signs:

```
:echo +123.4
" returns 123.4

:echo -1.234e2
" returns -123.4

:echo 0.25
" returns 0.25

:echo 2.5e-1
" returns 0.25
```

You need to give a float a dot and trailing digits. `25e-2` (no dot) and `1234.` (has a dot, but no trailing digits) are both invalid float numbers.

Float Arithmetic

When doing an arithmetic expression between a number and a float, Vim coerces the result to a float.

```
:echo 5 / 2.0
" returns 2.5
```

Float and float arithmetic gives you another float.

```
:echo 1.0 + 1.0
" returns 2.0
```

String

Strings are characters surrounded by either double-quotes (`""`) or single-quotes (`''`). “Hello”, “123”, and ‘123.4’ are examples of strings.

String Concatenation

To concatenate a string in Vim, use the `.` operator.

```
:echo "Hello" . " world"
" returns "Hello world"
```

String Arithmetic

When you run arithmetic operators (`+ - * /`) with a number and a string, Vim coerces the string into a number.

```
:echo "12 donuts" + 3
" returns 15
```

When Vim sees “12 donuts”, it extracts the 12 from the string and converts it into the number 12. Then it performs addition, returning 15. For this string-to-number coercion to work, the number character needs to be the *first character* in the string.

The following won’t work because 12 is not the first character in the string:

```
:echo "donuts 12" + 3
" returns 3
```

This also won’t work because an empty space is the first character of the string:

```
:echo " 12 donuts" + 3
" returns 3
```

This coercion works even with two strings:

```
:echo "12 donuts" + "6 pastries"
" returns 18
```

This works with any arithmetic operator, not only `+`:

```
:echo "12 donuts" * "5 boxes"
" returns 60

:echo "12 donuts" - 5
" returns 7

:echo "12 donuts" / "3 people"
" returns 4
```

A neat trick to force a string-to-number conversion is to just add `0` or multiply by `1`:

```
:echo "12" + 0
" returns 12

:echo "12" * 1
" returns 12
```

When arithmetic is done against a float in a string, Vim treats it like an integer, not a float:

```
:echo "12.0 donuts" + 12
" returns 24, not 24.0
```

Number and String Concatenation

You can coerce a number into a string with a dot operator (`.`):

```
:echo 12 . "donuts"
" returns "12donuts"
```

The coercion only works with number data type, not float. This won't work:

```
:echo 12.0 . "donuts"
" does not return "12.0donuts" but throws an error
```

String Conditionals

Recall that 0 is falsy and all non-0 numbers are truthy. This is also true when using string as conditionals.

In the following if statement, Vim coerces "12donuts" into 12, which is truthy:

```
:if "12donuts"
:  echo "Yum"
:endif
" returns "Yum"
```

On the other hand, this is falsy:

```
:if "donuts12"
:  echo "Nope"
:endif
" rerturns nothing
```

Vim coerces "donuts12" into 0, because the first character is not a number.

Double vs Single quotes

Double quotes behave differently than single quotes. Single quotes display characters literally while double quotes accept special characters.

What are special characters? Check out the newline and double-quotes display:

```
:echo "hello\nworld"
" returns
" hello
" world

:echo "hello \"world\""
" returns "hello "world""
```

Compare that with single-quotes:

```
:echo 'hello\nworld'
" returns 'hello\nworld'

:echo 'hello \"world\"'
" returns 'hello \"world\"'
```

Special characters are special string characters that when escaped, behave differently. `\n` acts like a newline. `\"` behaves like a literal `"`. For a list of other special characters, check out `:h expr-quote`.

String Procedures

Let's look at some built-in string procedures.

You can get the length of a string with `strlen()`.

```
:echo strlen("choco")
" returns 5
```

You can convert string to a number with `str2nr()`:

```
:echo str2nr("12donuts")
" returns 12

:echo str2nr("donuts12")
" returns 0
```

Similar to the string-to-number coercion earlier, if the number is not the first character, Vim won't catch it.

The good news is that Vim has a method that transforms a string to a float, `str2float()`:

```
:echo str2float("12.5donuts")
" returns 12.5
```

You can substitute a pattern in a string with the `substitute()` method:

```
:echo substitute("sweet", "e", "o", "g")
" returns "swoot"
```

The last parameter, "g", is the global flag. With it, Vim will substitute all matching occurrences. Without it, Vim will only substitute the first match.

```
:echo substitute("sweet", "e", "o", "")
" returns "swoet"
```

The substitute command can be combined with `getline()`. Recall that the function `getline()` gets the text on the given line number. Suppose you have the text "chocolate donut" on line `5`. You can use the procedure:

```
:echo substitute(getline(5), "chocolate", "glazed", "g")
" returns glazed donut
```

There are many other string procedures. Check out `:h string-functions`.

List

A Vimscript list is like an Array in Javascript or List in Python. It is an *ordered* sequence of items. You can mix-and-match the content with different data types:

```
[1,2,3]
['a', 'b', 'c']
[1,'a', 3.14]
[1,2,[3,4]]
```

Sublists

Vim list is zero-indexed. You can access a particular item in a list with `[n]`, where n is the index.

```
:echo ["a", "sweet", "dessert"][0]
" returns "a"

:echo ["a", "sweet", "dessert"][2]
" returns "dessert"
```

If you go over the maximum index number, Vim will throw an error saying that the index is out of range:

```
:echo ["a", "sweet", "dessert"][999]
" returns an error
```

When you go below zero, Vim will start the index from the last element. Going past the minimum index number will also throw you an error:

```
:echo ["a", "sweet", "dessert"][-1]
" returns "dessert"

:echo ["a", "sweet", "dessert"][-3]
" returns "a"

:echo ["a", "sweet", "dessert"][-999]
" returns an error
```

You can "slice" several elements from a list with `[n:m]`, where `n` is the starting index and `m` is the ending index.

```
:echo ["chocolate", "glazed", "plain", "strawberry", "lemon", "sugar", "cream"][2:4]
" returns ["plain", "strawberry", "lemon"]
```

If you don't pass `m` (`[n:]`), Vim will return the rest of the elements starting from the nth element. If you don't pass `n` (`[:m]`), Vim will return the first element up to the mth element.

```
:echo ["chocolate", "glazed", "plain", "strawberry", "lemon", "sugar", "cream"][2:]
" returns ['plain', 'strawberry', 'lemon', 'sugar', 'cream']

:echo ["chocolate", "glazed", "plain", "strawberry", "lemon", "sugar", "cream"][:4]
" returns ['chocolate', 'glazed', 'plain', 'strawberry', 'lemon']
```

You can pass an index that exceeds the maximum items when slicing an array.

```
:echo ["chocolate", "glazed", "plain", "strawberry", "lemon", "sugar", "cream"][2:99\
9]
" returns ['plain', 'strawberry', 'lemon', 'sugar', 'cream']
```

Slicing String

You can slice and target strings just like lists:

```
:echo "choco"[0]
" returns "c"

:echo "choco"[1:3]
" returns "hoc"

:echo "choco"[:3]
" returns choc

:echo "choco"[1:]
" returns hoco
```

List Arithmetic

You can use + to concatenate and mutate a list:

```
:let sweetList = ["chocolate", "strawberry"]
:let sweetList += ["sugar"]
:echo sweetList
" returns ["chocolate", "strawberry", "sugar"]
```

List Functions

Let's explore Vim's built-in list functions.

To get the length of a list, use `len()`:

```
:echo len(["chocolate", "strawberry"])
" returns 2
```

To prepend an element to a list, you can use `insert()`:

```
:let sweetList = ["chocolate", "strawberry"]
:call insert(sweetList, "glazed")

:echo sweetList
" returns ["glazed", "chocolate", "strawberry"]
```

You can also pass `insert()` the index where you want to prepend the element to. If you want to prepend an item before the second element (index 1):

```
:let sweeterList = ["glazed", "chocolate", "strawberry"]
:call insert(sweeterList, "cream", 1)

:echo sweeterList
" returns ['glazed', 'cream', 'chocolate', 'strawberry']
```

To remove a list item, use `remove()`. It accepts a list and the element index you want to remove.

```
:let sweeterList = ["glazed", "chocolate", "strawberry"]
:call remove(sweeterList, 1)

:echo sweeterList
" returns ['glazed', 'strawberry']
```

You can use `map()` and `filter()` on a list. To filter out element containing the phrase "choco":

```
:let sweeterList = ["glazed", "chocolate", "strawberry"]
:call filter(sweeterList, 'v:val !~ "choco"')
:echo sweeterList
" returns ["glazed", "strawberry"]

:let sweetestList = ["chocolate", "glazed", "sugar"]
:call map(sweetestList, 'v:val . " donut"')
:echo sweetestList
" returns ['chocolate donut', 'glazed donut', 'sugar donut']
```

The `v:val` variable is a Vim special variable. It is available when iterating a list or a dictionary using `map()` or `filter()`. It represents each iterated item.

For more, check out `:h list-functions`.

List Unpacking

You can unpack a list and assign variables to the list items:

```
:let favoriteFlavor = ["chocolate", "glazed", "plain"]
:let [flavor1, flavor2, flavor3] = favoriteFlavor

:echo flavor1
" returns "chocolate"

:echo flavor2
" returns "glazed"
```

To assign the rest of list items, you can use `;` followed with a variable name:

```
:let favoriteFruits = ["apple", "banana", "lemon", "blueberry", "raspberry"]
:let [fruit1, fruit2; restFruits] = favoriteFruits

:echo fruit1
" returns "apple"

:echo restFruits
" returns ['lemon', 'blueberry', 'raspberry']
```

Modifying List

You can modify a list item directly:

```
:let favoriteFlavor = ["chocolate", "glazed", "plain"]
:let favoriteFlavor[0] = "sugar"
:echo favoriteFlavor
" returns ['sugar', 'glazed', 'plain']
```

You can mutate multiple list items directly:

```
:let favoriteFlavor = ["chocolate", "glazed", "plain"]
:let favoriteFlavor[2:] = ["strawberry", "chocolate"]
:echo favoriteFlavor
returns ['chocolate', 'glazed', 'strawberry', 'chocolate']
```

Dictionary

A Vimscript dictionary is an associative, unordered list. A non-empty dictionary consists of at least a key-value pair.

```
{"breakfast": "waffles", "lunch": "pancakes"}
{"meal": ["breakfast", "second breakfast", "third breakfast"]}
{"dinner": 1, "dessert": 2}
```

A Vim dictionary data object uses string for key. If you try to use a number, Vim will coerce it into a string.

```
:let breakfastNo = {1: "7am", 2: "9am", "11ses": "11am"}

:echo breakfastNo
" returns {'1': '7am', '2': '9am', '11ses': '11am'}
```

If you are too lazy to put quotes around each key, you can use the `#{}` notation:

```
:let mealPlans = #{breakfast: "waffles", lunch: "pancakes", dinner: "donuts"}

:echo mealPlans
" returns {'lunch': 'pancakes', 'breakfast': 'waffles', 'dinner': 'donuts'}
```

The only requirement for using the `#{}` syntax is that each key must be either:

- ASCII character.
- Digit.
- An underscore (_).
- A hyphen (-).

Just like list, you can use any data type as values.

```
:let mealPlan = {"breakfast": ["pancake", "waffle", "hash brown"], "lunch": WhatsFor\
Lunch(), "dinner": {"appetizer": "gruel", "entree": "more gruel"}}
```

Accessing Dictionary

To access a value from a dictionary, you can call the key with either the square brackets (`['key']`) or the dot notation (`.key`).

```
:let meal = {"breakfast": "gruel omelettes", "lunch": "gruel sandwiches", "dinner": \
"more gruel"}

:let breakfast = meal['breakfast']
:let lunch = meal.lunch

:echo breakfast
" returns "gruel omelettes"

:echo lunch
" returns "gruel sandwiches"
```

Modifying Dictionary

You can modify or even add a dictionary content:

```
:let meal = {"breakfast": "gruel omelettes", "lunch": "gruel sandwiches"}

:let meal.breakfast = "breakfast tacos"
:let meal["lunch"] = "tacos al pastor"
:let meal["dinner"] = "quesadillas"

:echo meal
" returns {'lunch': 'tacos al pastor', 'breakfast': 'breakfast tacos', 'dinner': 'qu\
esadillas'}
```

Dictionary Functions

Let's explore some of Vim's built-in functions to handle Dictionaries.

To check the length of a dictionary, use `len()`.

```
:let mealPlans = #{breakfast: "waffles", lunch: "pancakes", dinner: "donuts"}

:echo len(meaPlans)
" returns 3
```

To see if a dictionary contains a specific key, use `has_key()`

```
:let mealPlans = #{breakfast: "waffles", lunch: "pancakes", dinner: "donuts"}

:echo has_key(mealPlans, "breakfast")
" returns 1

:echo has_key(mealPlans, "dessert")
" returns 0
```

To see if a dictionary has any item, use `empty()`. The `empty()` procedure works with all data types: list, dictionary, string, number, float, etc.

```
:let mealPlans = #{breakfast: "waffles", lunch: "pancakes", dinner: "donuts"}
:let noMealPlan = {}

:echo empty(noMealPlan)
" returns 1

:echo empty(mealPlans)
" returns 0
```

To remove an entry from a dictionary, use `remove()`.

```
:let mealPlans = #{breakfast: "waffles", lunch: "pancakes", dinner: "donuts"}

:echo "removing breakfast: " . remove(mealPlans, "breakfast")
" returns "removing breakfast: 'waffles'""

:echo mealPlans
" returns {'lunch': 'pancakes', 'dinner': 'donuts'}
```

To convert a dictionary into a list of lists, use `items()`:

```
:let mealPlans = #{breakfast: "waffles", lunch: "pancakes", dinner: "donuts"}

:echo items(mealPlans)
" returns [['lunch', 'pancakes'], ['breakfast', 'waffles'], ['dinner', 'donuts']]
```

`filter()` and `map()` are also available.

```
:let breakfastNo = {1: "7am", 2: "9am", "11ses": "11am"}
:call filter(breakfastNo, 'v:key > 1')

:echo breakfastNo
" returns {'2': '9am', '11ses': '11am'}

:let mealPlans = #{breakfast: "waffles", lunch: "pancakes", dinner: "donuts"}
:call map(mealPlans, 'v:key . " and milk"')

:echo mealPlans
" returns {'lunch': 'lunch and milk', 'breakfast': 'breakfast and milk', 'dinner': '\
dinner and milk'}
```

The `v:key` is Vim's special variable, much like `v:val`. When iterating through a dictionary, `v:key` will hold the value of the current iterated key.

To see more dictionary functions, check out `:h dict-functions`.

Special Primitives

Vim has special primitives:

- `v:false`
- `v:true`
- `v:none`
- `v:null`

By the way, `v:` is Vim's built-in variable. They will be covered more in a later chapter.

In my experience, you won't use these special primitives often. If you need a truthy / falsy value, you can just use `0` (falsy) and non-`0` (truthy). If you need an empty string, just use `""`. But it is still good to know, so let's quickly go over them.

True

This is equivalent to `true`. It is equivalent to a number with value of non-`0` . When decoding json with `json_encode()`, it is interpreted as "true".

```
:echo json_encode({"test": v:true})
" returns {"test": true}
```

False

This is equivalent to `false`. It is equivalent to a number with value of `0`. When decoding json with `json_encode()`, it is interpreted as "false".

```
:echo json_encode({"test": v:false})
" returns {"test": false}
```

None

It is equivalent to an empty string. When decoding json with `json_encode()`, it is interpreted as an empty item (`null`).

```
:echo json_encode({"test": v:none})
" returns {"test": null}
```

Null

Similar to `v:none`.

```
:echo json_encode({"test": v:null})
" returns {"test": null}
```

Learn Data Types The Smart Way

In this chapter, you learned about Vimscript's basic data types: number, float, string, list, dictionary, and special. Learning these is the first step to start Vimscript programming.

In the next chapter, you will learn how to combine them for writing expressions like equalities, conditionals, and loops.

Ch25. Vimscript Conditionals And Loops

After learning what the basic data types are, the next step is to learn how to combine them together to start writing a basic program. A basic program consists of conditionals and loops.

In this chapter, you will learn how to use Vimscript data types to write conditionals and loops.

Relational Operators

Vimscript relational operators are similar to many programming languages:

```
a == b              equal to
a != b              not equal to
a >  b              greater than
a >= b              greater than or equal to
a <  b              less than
a <= b              less than or equal to
```

For example:

```
:echo 5 == 5
:echo 5 != 5
:echo 10 > 5
:echo 10 >= 5
:echo 10 < 5
:echo 5 <= 5
```

Recall that strings are coerced into numbers in an arithmetic expression. Here Vim also coerces strings into numbers in an equality expression. "5foo" is coerced into 5 (truthy):

```
:echo 5 == "5foo"
" returns true
```

Also recall that if you start a string with a non-numerical character like "foo5", the string is converted into number 0 (falsy).

```
echo 5 == "foo5"
" returns false
```

String Logic Operators

Vim has more relational operators for comparing strings:

```
a =~ b
a !~ b
```

For examples:

```
let str = "hearty breakfast"

echo str =~ "hearty"
" returns true

echo str =~ "dinner"
" returns false

echo str !~ "dinner"
" returns true
```

The =~ operator performs a regex match against the given string. In the example above, `str =~ "hearty"` returns true because `str` *contains* the "hearty" pattern. You can always use == and !=, but using them will compare the expression against the entire string. =~ and !~ are more flexible choices.

```
echo str == "hearty"
" returns false

echo str == "hearty breakfast"
" returns true
```

Let's try this one. Note the uppercase "H":

```
echo str =~ "Hearty"
" true
```

It returns true even though "Hearty" is capitalized. Interesting… It turns out that my Vim setting is set to ignore case (`set ignorecase`), so when Vim checks for equality, it uses my Vim setting and ignores the case. If I were to turn off ignore case (`set noignorecase`), the comparison now returns false.

```
set noignorecase
echo str =~ "Hearty"
" returns false because case matters

set ignorecase
echo str =~ "Hearty"
" returns true because case doesn't matter
```

If you are writing a plugin for others, this is a tricky situation. Does the user use `ignorecase` or `noignorecase`? You definitely do *not* want to force your users to change their ignore case option. So what do you do?

Luckily, Vim has an operator that can *always* ignore or match case. To always match case, add a `#` at the end.

```
set ignorecase
echo str =~# "hearty"
" returns true

echo str =~# "HearTY"
" returns false

set noignorecase
echo str =~# "hearty"
" true

echo str =~# "HearTY"
" false

echo str !~# "HearTY"
" true
```

To always ignore case when comparing, append it with `?`:

```
set ignorecase
echo str =~? "hearty"
" true

echo str =~? "HearTY"
" true

set noignorecase
echo str =~? "hearty"
```

```
" true

echo str =~? "HearTY"
" true

echo str !~? "HearTY"
" false
```

I prefer to use `#` to always match the case and be on the safe side.

If

Now that you have seen Vim's equality expressions, let's touch a fundamental conditional operator, the `if` statement.

At minimum, the syntax is:

```
if {clause}
  {some expression}
endif
```

You can extend the case analysis with `elseif` and `else`.

```
if {predicate1}
  {expression1}
elseif {predicate2}
  {expression2}
elseif {predicate3}
  {expression3}
else
  {expression4}
endif
```

For example, the plugin vim-signify[61] uses a different installation method depending on your Vim settings. Below is the installation instruction from their `readme`, using the `if` statement:

[61] https://github.com/mhinz/vim-signify

```
if has('nvim') || has('patch-8.0.902')
  Plug 'mhinz/vim-signify'
else
  Plug 'mhinz/vim-signify', { 'branch': 'legacy' }
endif
```

Ternary Expression

Vim has a ternary expression for a one-liner case analysis:

```
{predicate} ? expressiontrue : expressionfalse
```

For example:

```
echo 1 ? "I am true" : "I am false"
```

Since `1` is truthy, Vim echoes "I am true". Suppose you want to conditionally set the `background` to dark if you are using Vim past a certain hour. Add this to vimrc:

```
let &background = strftime("%H") < 18 ? "light" : "dark"
```

`&background` is the `'background'` option in Vim. `strftime("%H")` returns the current time in hours. If it is not yet 6 PM, use a light background. Otherwise, use a dark background.

Or

The logical "or" (`||`) works like many programming languages.

```
{Falsy expression}  || {Falsy expression}   false
{Falsy expression}  || {Truthy expression}  true
{Truthy expression} || {Falsy expression}   true
{Truthy expression} || {Truthy expression}  true
```

Vim evaluates the expression and return either `1` (truthy) or `0` (falsy).

```
echo 5 || 0
" returns 1

echo 5 || 5
" returns 1

echo 0 || 0
" returns 0

echo "foo5" || "foo5"
" returns 0

echo "5foo" || "foo5"
" returns 1
```

If the current expression evaluates to truthy, the subsequent expression won't be evaluated.

```
let one_dozen = 12

echo one_dozen || two_dozen
" returns 1

echo two_dozen || one_dozen
" returns error
```

Note that `two_dozen` is never defined. The expression `one_dozen || two_dozen` doesn't throw any error because `one_dozen` is evaluated first found to be truthy, so Vim doesn't evaluate `two_dozen`.

And

The logical "and" (`&&`) is the complement of the logical or.

```
{Falsy Expression}  && {Falsy Expression}   false
{Falsy expression}  && {Truthy expression}  false
{Truthy Expression} && {Falsy Expression}   false
{Truthy expression} && {Truthy expression}  true
```

For example:

```
echo 0 && 0
" returns 0

echo 0 && 10
" returns 0
```

Unlike “or”, “and” will evaluate the subsequent expression after it reaches the first falsy expression. It will continue to evaluate the subsequent truthy expressions until the end or when it sees the first falsy expression.

```
let one_dozen = 12
echo one_dozen && 10
" returns 1

echo one_dozen && v:false
" returns 0

echo one_dozen && two_dozen
" returns error

echo exists("one_dozen") && one_dozen == 12
" returns 1
```

For

The `for` loop is commonly used with the list data type.

```
let breakfasts = ["pancakes", "waffles", "eggs"]

for breakfast in breakfasts
  echo breakfast
endfor
```

It works with nested list:

```
let meals = [["breakfast", "pancakes"], ["lunch", "fish"], ["dinner", "pasta"]]

for [meal_type, food] in meals
  echo "I am having " . food . " for " . meal_type
endfor
```

You can technically use the `for` loop with a dictionary using the `keys()` method.

```
let beverages = #{breakfast: "milk", lunch: "orange juice", dinner: "water"}
for beverage_type in keys(beverages)
  echo "I am drinking " . beverages[beverage_type] . " for " . beverage_type
endfor
```

While

Another common loop is the `while` loop.

```
let counter = 1
while counter < 5
  echo "Counter is: " . counter
  let counter += 1
endwhile
```

To get the content of the current line to the last line:

```
let current_line = line(".")
let last_line = line("$")

while current_line <= last_line
  echo getline(current_line)
  let current_line += 1
endwhile
```

Error Handling

Often your program doesn't run the way you expect it to. As a result, it throws you for a loop (pun intended). What you need is a proper error handling.

Break

When you use `break` inside a `while` or `for` loop, it stops the loop.

To get the texts from the start of the file to the current line, but stop when you see the word "donut":

```
let line = 0
let last_line = line("$")
let total_word = ""

while line <= last_line
  let line += 1
  let line_text = getline(line)
  if line_text =~# "donut"
    break
  endif
  echo line_text
  let total_word .= line_text . " "
endwhile

echo total_word
```

If you have the text:

```
one
two
three
donut
four
five
```

Running the above `while` loop gives “one two three” and not the rest of the text because the loop breaks once it matches “donut”.

Continue

The `continue` method is similar to `break`, where it is invoked during a loop. The difference is that instead of breaking out of the loop, it just skips that current iteration.

Suppose you have the same text but instead of `break`, you use `continue`:

```
let line = 0
let last_line = line("$")
let total_word = ""

while line <= last_line
  let line += 1
  let line_text = getline(line)
  if line_text =~# "donut"
    continue
  endif
  echo line_text
  let total_word .= line_text . " "
endwhile

echo total_word
```

This time it returns `one two three four five`. It skips the line with the word "donut", but the loop continues.

Try, Finally, And Catch

Vim has a `try`, `finally`, and `catch` to handle errors. To simulate an error, you can use the `throw` command.

```
try
  echo "Try"
  throw "Nope"
endtry
```

Run this. Vim will complain with `"Exception not caught: Nope` error.

Now add a catch block:

```
try
  echo "Try"
  throw "Nope"
catch
  echo "Caught it"
endtry
```

Now there is no longer any error. You should see "Try" and "Caught it" displayed.

Let's remove the `catch` and add a `finally`:

```
try
  echo "Try"
  throw "Nope"
  echo "You won't see me"
finally
  echo "Finally"
endtry
```

Run this. Now Vim displays the error and “Finally”.

Let’s put all of them together:

```
try
  echo "Try"
  throw "Nope"
catch
  echo "Caught it"
finally
  echo "Finally"
endtry
```

This time Vim displays both “Caught it” and “Finally”. No error is displayed because Vim caught it.

Errors come from different places. Another source of error is calling a nonexistent function, like `Nope()` below:

```
try
  echo "Try"
  call Nope()
catch
  echo "Caught it"
finally
  echo "Finally"
endtry
```

The difference between `catch` and `finally` is that `finally` is always run, error or not, where a catch is only run when your code gets an error.

You can catch specific error with `:catch`. According to `:h :catch`:

```
catch /^Vim:Interrupt$/.              " catch interrupts (CTRL-C)
catch /^Vim\\%((\\a\\+)\\)\\=:E/.     " catch all Vim errors
catch /^Vim\\%((\\a\\+)\\)\\=:/.      " catch errors and interrupts
catch /^Vim(write):/.                 " catch all errors in :write
catch /^Vim\\%((\\a\\+)\\)\\=:E123:/ " catch error E123
catch /my-exception/.                 " catch user exception
catch /.*/                            " catch everything
catch.                                " same as /.*/
```

Inside a `try` block, an interrupt is considered a catchable error.

```
try
  catch /^Vim:Interrupt$/
  sleep 100
endtry
```

In your vimrc, if you use a custom colorscheme, like gruvbox[62], and you accidentally delete the colorscheme directory but still have the line `colorscheme gruvbox` in your vimrc, Vim will throw an error when you `source` it. To fix this, I added this in my vimrc:

```
try
  colorscheme gruvbox
catch
  colorscheme default
endtry
```

Now if you `source` vimrc without `gruvbox` directory, Vim will use the `colorscheme default`.

Learn conditionals the smart way

In the previous chapter, you learned about Vim basic data types. In this chapter, you learned how to combine them to write basic programs using conditionals and loops. These are the building blocks of programming.

Next, let's learn about variable scopes.

[62] https://github.com/morhetz/gruvbox

Ch26. Vimscript Variables And Scopes

Before diving into Vimscript functions, let's learn about the different sources and scopes of Vim variables.

Mutable And Immutable Variables

You can assign a value to a variable in Vim with `let`:

```
let pancake = "pancake"
```

Later you can call that variable any time.

```
echo pancake
" returns "pancake"
```

`let` is mutable, meaning you can change the value at any time in the future.

```
let pancake = "pancake"
let pancake = "not waffles"

echo pancake
" returns "not waffles"
```

Notice that when you want to change the value of a set variable, you still need to use `let`.

```
let beverage = "milk"

beverage = "orange juice"
" throws an error
```

You can define an immutable variable with `const`. Being immutable, once a variable value is assigned, you cannot reassign it with a different value.

```
const waffle = "waffle"
const waffle = "pancake"
" throws an error
```

Variable Sources

There are three sources for variables: environment variable, option variable, and register variable.

Environment Variable

Vim can access your terminal environment variable. For example, if you have the `SHELL` environment variable available in your terminal, you can access it from Vim with:

```
echo $SHELL
" returns $SHELL value. In my case, it returns /bin/bash
```

Option Variable

You can access Vim options with `&` (these are the settings you access with `set`).

For example, to see what background Vim uses, you can run:

```
echo &background
" returns either "light" or "dark"
```

Alternatively, you can always run `set background?` to see the value of the `background` option.

Register Variable

You can access Vim registers (Ch. 08) with `@`.

Suppose the value “chocolate” is already saved in register a. To access it, you can use `@a`. You can also update it with `let`.

```
echo @a
" returns chocolate

let @a .= " donut"

echo @a
" returns "chocolate donut"
```

Now when you paste from register `a` (`"ap`), it will return "chocolate donut". The operator `.=` concatenates two strings. The expression `let @a .= " donut"` is the same as `let @a = @a . " donut"`

Variable Scopes

There are 9 different variable scopes in Vim. You can recognize them from their prepended letter:

```
g:          Global variable
{nothing}   Global variable
b:          Buffer-local variable
w:          Window-local variable
t:          Tab-local variable
s:          Sourced Vimscript variable
l:          Function local variable
a:          Function formal parameter variable
v:          Built-in Vim variable
```

Global variable

When you are declaring a "regular" variable:

```
let pancake = "pancake"
```

`pancake` is actually a global variable. When you define a global variable, you can call them from anywhere.

Prepending `g:` to a variable also creates a global variable.

```
let g:waffle = "waffle"
```

In this case both `pancake` and `g:waffle` have the same scope. You can call each of them with or without `g:`.

```
echo pancake
" returns "pancake"

echo g:pancake
"returns "pancake"

echo waffle
" returns "waffle"

echo g:waffle
" returns "waffle"
```

Buffer Variable

A variable preceded with `b:` is a buffer variable. A buffer variable is a variable that is local to the current buffer (Ch. 02). If you have multiple buffers open, each buffer will have their own separate list of buffer variables.

In buffer 1:

```
const b:donut = "chocolate donut"
```

In buffer 2:

```
const b:donut = "blueberry donut"
```

If you run `echo b:donut` from buffer 1, it will return "chocolate donut". If you run it from buffer 2, it will return "blueberry donut".

On the side note, Vim has a *special* buffer variable `b:changedtick` that keeps track of all the changes done to the current buffer.

1. Run `echo b:changedtick` and note the number it returns..
2. Make changes in Vim.
3. Run `echo b:changedtick` again and note the number it now returns.

Window Variable

A variable preceded with `w:` is a window variable. It exists only in that window.

In window 1:

```
const w:donut = "chocolate donut"
```

In window 2:

```
const w:donut = "raspberry donut"
```

On each window, you can call `echo w:donut` to get unique values.

Tab Variable

A variable preceded with `t:` is a tab variable. It exists only in that tab.

In tab 1:

```
const t:donut = "chocolate donut"
```

In tab 2:

```
const t:donut = "blackberry donut"
```

On each tab, you can call `echo t:donut` to get unique values.

Script variable

A variable preceded with `s:` is a script variable. These variables can only be accessed from inside that script.

If you have an arbitrary file `dozen.vim` and inside it you have:

```
let s:dozen = 12

function Consume()
  let s:dozen -= 1
  echo s:dozen " is left"
endfunction
```

Source the file with `:source dozen.vim`. Now call the `Consume` function:

```
:call Consume()
" returns "11 is left"

:call Consume()
" returns "10 is left"

:echo s:dozen
" Undefined variable error
```

When you call `Consume`, you see it decrements the `s:dozen` value as expected. When you try to get `s:dozen` value directly, Vim won't find it because you are out of scope. `s:dozen` is only accessible from inside `dozen.vim`.

Each time you source the `dozen.vim` file, it resets the `s:dozen` counter. If you are in the middle of decrementing `s:dozen` value and you run `:source dozen.vim`, the counter resets back to 12. This can be a problem for unsuspecting users. To fix this issue, refactor the code:

```
if !exists("s:dozen")
  let s:dozen = 12
endif

function Consume()
  let s:dozen -= 1
  echo s:dozen
endfunction
```

Now when you source `dozen.vim` while in the middle of decrementing, Vim reads `!exists("s:dozen")`, finds that it is true, and doesn't reset the value back to 12.

Function Local And Function Formal Parameter variable

Both the function local variable (`l:`) and the function formal variable (`a:`) will be covered in the next chapter.

Built-in Vim Variables

A variable prepended with `v:` is a special built-in Vim variable. You cannot define these variables. You have seen some of them already.

- `v:version` tells you what Vim version you are using.
- `v:key` contains the current item value when iterating through a dictionary.
- `v:val` contains the current item value when running a `map()` or `filter()` operation.
- `v:true`, `v:false`, `v:null`, and `v:none` are special data types.

There are other variables. For a list of Vim built-in variables, check out `:h vim-variable` or `:h v:`.

Using Vim Variable Scopes The Smart Way

Being able to quickly access environment, option, and register variables give you a broad flexibility to customize your editor and terminal environment. You also learned that Vim has 9 different variable scopes, each existing under a certain constraints. You can take advantage of these unique variable types to decouple your program.

You made it this far. You learned about data types, means of combinations, and variable scopes. Only one thing is left: functions.

Ch27. Vimscript Functions

Functions are means of abstraction, the third element in learning a new language.

In the previous chapters, you have seen Vimscript native functions (`len()`, `filter()`, `map()`, etc.) and custom functions in action. In this chapter, you will go deeper to learn how functions work.

Function Syntax Rules

At the core, a Vimscript function has the following syntax:

```
function {FunctionName}()
  {do-something}
endfunction
```

A function definition must start with a capital letter. It starts with the `function` keyword and ends with `endfunction`. Below is a valid function:

```
function! Tasty()
  echo "Tasty"
endfunction
```

The following is not a valid function because it does not start with a capital letter.

```
function tasty()
  echo "Tasty"
endfunction
```

If you prepend a function with the script variable (`s:`), you can use it with a lower case. `function s:tasty()` is a valid name. The reason why Vim requires you to use an uppercase name is to prevent confusion with Vim's built-in functions (all lowercase).

A function name cannot start with a number. `1Tasty()` is not a valid function name, but `Tasty1()` is. A function also cannot contain non-alphanumeric characters besides _. `Tasty-food()`, `Tasty&food()`, and `Tasty.food()` are not valid function names. `Tasty_food()` *is*.

If you define two functions with the same name, Vim will throw an error complaining that the function `Tasty` already exists. To overwrite the previous function with the same name, add a `!` after the `function` keyword.

```
function! Tasty()
  echo "Tasty"
endfunction
```

Listing Available Functions

To see all the built-in and custom functions in Vim, you can run `:function` command. To look at the content of the `Tasty` function, you can run `:function Tasty`.

You can also search for functions with pattern with `:function /pattern`, similar to Vim's search navigation (`/pattern`). To search for all function containing the phrase "map", run `:function /map`. If you use external plugins, Vim will display the functions defined in those plugins.

If you want to look at where a function originates, you can use the `:verbose` command with the `:function` command. To look at where all the functions containing teh word "map" are originated, run:

```
:verbose function /map
```

When I ran it, I got a number of results. This one tells me that the function `fzf#vim#maps` autoload function (to recap, refer to Ch. 23) is written inside `~/.vim/plugged/fzf.vim/autoload/fzf/vim.vim` file, on line 1263. This is useful for debugging.

```
function fzf#vim#maps(mode, ...)
        Last set from ~/.vim/plugged/fzf.vim/autoload/fzf/vim.vim line 1263
```

Removing A Function

To remove an existing function, use `:delfunction {function-name}`. To delete `Tasty`, run `:delfunction Tasty`.

Function Return Value

For a function to return a value, you need to pass it an explicit `return` value. Otherwise, Vim automatically returns an implicit value of `0`.

```
function! Tasty()
  echo "Tasty"
endfunction
```

An empty `return` is also equivalent to a `0` value.

```
function! Tasty()
  echo "Tasty"
  return
endfunction
```

If you run `:echo Tasty()` using the function above, after Vim displays "Tasty", it returns `0`, the implicit return value. To make `Tasty()` to return "Tasty" value, you can do this:

```
function! Tasty()
  return "Tasty"
endfunction
```

Now when you run `:echo Tasty()`, it returns "Tasty" string.

You can use a function inside an expression. Vim will use the return value of that function. The expression `:echo Tasty() . " Food!"` outputs "Tasty Food!"

Formal Arguments

To pass a formal argument `food` to your `Tasty` function, you can do this:

```
function! Tasty(food)
  return "Tasty " . a:food
endfunction

echo Tasty("pastry")
" returns "Tasty pastry"
```

`a:` is one of the variable scopes mentioned in the last chapter. It is the formal parameter variable. It is Vim's way to get a formal parameter value in a function. Without it, Vim will throw an error:

```
function! Tasty(food)
  return "Tasty " . food
endfunction

echo Tasty("pasta")
" returns "undefined variable name" error
```

Function Local Variable

Let's address the other variable you didn't learn on the previous chapter: the function local variable (`l:`).

When writing a function, you can define a variable inside:

```
function! Yummy()
  let location = "tummy"
  return "Yummy in my " . location
endfunction

echo Yummy()
" returns "Yummy in my tummy"
```

In this context, the variable `location` is the same as `l:location`. When you define a variable in a function, that variable is *local* to that function. When a user sees `location`, it could easily be mistaken as a global variable. I prefer to be more verbose than not, so I prefer to put `l:` to indicate that this is a function variable.

Another reason to use `l:count` is that Vim has special variables with aliases that look like regular variables. `v:count` is one example. It has an alias of `count`. In Vim, calling `count` is the same as calling `v:count`. It is easy to accidentally call one of those special variables.

```
function! Calories()
  let count = "count"
  return "I do not " . count . " my calories"
endfunction

echo Calories()
" throws an error
```

The execution above throws an error because `let count = "Count"` implicitly attempts to redefine Vim's special variable `v:count`. Recall that special variables (`v:`) are read-only. You cannot mutate it. To fix it, use `l:count`:

```
function! Calories()
  let l:count = "count"
  return "I do not " . l:count . " my calories"
endfunction

echo Calories()
" returns "I do not count my calories"
```

Calling A Function

Vim has a `:call` command to call a function.

```
function! Tasty(food)
  return "Tasty " . a:food
endfunction

call Tasty("gravy")
```

The `call` command does not output the return value. Let's call it with `echo`.

```
echo call Tasty("gravy")
```

Woops, you get an error. The `call` command above is a command-line command (`:call`). The `echo` command above is also a command-line command (`:echo`). You cannot call a command-line command with another command-line command. Let's try a different flavor of the `call` command:

```
echo call("Tasty", ["gravy"])
" returns "Tasty gravy"
```

To clear any confusion, you have just used two different `call` commands: the `:call` command-line command and the `call()` function. The `call()` function accepts as its first argument the function name (string) and its second argument the formal parameters (list).

To learn more about `:call` and `call()`, check out `:h call()` and `:h :call`.

Default Argument

You can provide a function parameter with a default value with `=`. If you call `Breakfast` with only one argument, the `beverage` argument will use the "milk" default value.

```
function! Breakfast(meal, beverage = "Milk")
  return "I had " . a:meal . " and " . a:beverage . " for breakfast"
endfunction

echo Breakfast("Hash Browns")
" returns hash browns and milk

echo Breakfast("Cereal", "Orange Juice")
" returns Cereal and Orange Juice
```

Variable Arguments

You can pass a variable argument with three-dots (. . .). Variable argument is useful when you don't know how many variables a user will give.

Suppose you are creating an all-you-can-eat buffet (you'll never know how much food your customer will eat):

```
function! Buffet(...)
  return a:1
endfunction
```

If you run `echo Buffet("Noodles")`, it will output "Noodles". Vim uses `a:1` to print the *first* argument passed to `...`, up to 20 (`a:1` is the first argument, `a:2` is the second argument, etc). If you run `echo Buffet("Noodles", "Sushi")`, it will still display just "Noodles", let's update it:

```
function! Buffet(...)
  return a:1 . " " . a:2
endfunction

echo Buffet("Noodles", "Sushi")
" Returns "Noodles Sushi"
```

The problem with this approach is if you now run `echo Buffet("Noodles")` (with only one variable), Vim complains that it has an undefined variable `a:2`. How can you make it flexible enough to display exactly what the user gives?

Luckily, Vim has a special variable `a:0` to display the *length* of the argument passed into `...`.

```
function! Buffet(...)
  return a:0
endfunction

echo Buffet("Noodles")
" returns 1

echo Buffet("Noodles", "Sushi")
" returns 2

echo Buffet("Noodles", "Sushi", "Ice cream", "Tofu", "Mochi")
" returns 5
```

With this, you can iterate using the length of the argument.

```
function! Buffet(...)
  let l:food_counter = 1
  let l:foods = ""
  while l:food_counter <= a:0
    let l:foods .= a:{l:food_counter} . " "
    let l:food_counter += 1
  endwhile
  return l:foods
endfunction
```

The curly braces `a:{l:food_counter}` is a string interpolation, it uses the value of `food_counter` counter to call the formal parameter arguments `a:1`, `a:2`, `a:3`, etc.

```
echo Buffet("Noodles")
" returns "Noodles"

echo Buffet("Noodles", "Sushi", "Ice cream", "Tofu", "Mochi")
" returns everything you passed: "Noodles Sushi Ice cream Tofu Mochi"
```

The variable argument has one more special variable: `a:000`. It has the value of all variable arguments in a list format.

```
function! Buffet(...)
  return a:000
endfunction

echo Buffet("Noodles")
" returns ["Noodles"]

echo Buffet("Noodles", "Sushi", "Ice cream", "Tofu", "Mochi")
" returns ["Noodles", "Sushi", "Ice cream", "Tofu", "Mochi"]
```

Let's refactor the function to use a `for` loop:

```
function! Buffet(...)
  let l:foods = ""
  for food_item in a:000
    let l:foods .= food_item . " "
  endfor
  return l:foods
endfunction

echo Buffet("Noodles", "Sushi", "Ice cream", "Tofu", "Mochi")
" returns Noodles Sushi Ice cream Tofu Mochi
```

Range

You can define a *ranged* Vimscript function by adding a `range` keyword at the end of the function definition. A ranged function has two special variables available: `a:firstline` and `a:lastline`.

```
function! Breakfast() range
  echo a:firstline
  echo a:lastline
endfunction
```

If you are on line 100 and you run `call Breakfast()`, it will display 100 for both `firstline` and `lastline`. If you visually highlight (`v`, `V`, or `Ctrl-V`) lines 101 to 105 and run `call Breakfast()`, `firstline` displays 101 and `lastline` displays 105. `firstline` and `lastline` displays the minimum and maximum range where the function is called.

You can also use `:call` and passing it a range. If you run `:11,20call Breakfast()`, it will display 11 for `firstline` and 20 for `lastline`.

You might ask, "That's nice that Vimscript function accepts range, but can't I get the line number with `line(".")`? Won't it do the same thing?"

Good question. If this is what you mean:

```
function! Breakfast()
  echo line(".")
endfunction
```

Calling `:11,20call Breakfast()` executes the `Breakfast` function 10 times (one for each line in the range). Compare that if you had passed the `range` argument:

```
function! Breakfast() range
  echo line(".")
endfunction
```

Calling `11,20call Breakfast()` executes the `Breakfast` function *once*.

If you pass a `range` keyword and you pass a numerical range (like `11,20`) on `call`, Vim only executes that function once. If you don't pass a `range` keyword and you pass a numerical range (like `11,20`) on `call`, Vim executes that function N times depending on the range (in this case, N = 10).

Dictionary

You can add a function as a dictionary item by adding a `dict` keyword when defining a function.

If you have a function `SecondBreakfast` that returns whatever `breakfast` item you have:

```
function! SecondBreakfast() dict
  return self.breakfast
endfunction
```

Let's add this function to the `meals` dictionary:

```
let meals = {"breakfast": "pancakes", "second_breakfast": function("SecondBreakfast"\
), "lunch": "pasta"}

echo meals.second_breakfast()
" returns "pancakes"
```

With `dict` keyword, the key variable `self` refers to the dictionary where the function is stored (in this case, the `meals` dictionary). The expression `self.breakfast` is equal to `meals.breakfast`.

An alternative way to add a function into a dictionary object to use a namespace.

```
function! meals.second_lunch()
  return self.lunch
endfunction

echo meals.second_lunch()
" returns "pasta"
```

With namespace, you do not have to use the `dict` keyword.

Funcref

A funcref is a reference to a function. It is one of Vimscript's basic data types mentioned in Ch. 24.

The expression `function("SecondBreakfast")` above is an example of funcref. Vim has a built-in function `function()` that returns a funcref when you pass it a function name (string).

```
function! Breakfast(item)
  return "I am having " . a:item . " for breakfast"
endfunction

let Breakfastify = Breakfast
" returns error

let Breakfastify = function("Breakfast")

echo Breakfastify("oatmeal")
" returns "I am having oatmeal for breakfast"

echo Breakfastify("pancake")
" returns "I am having pancake for breakfast"
```

In Vim, if you want to assign a function to a variable, you can't just run assign it directly like `let MyVar = MyFunc`. You need to use the `function()` function, like `let MyFar = function("MyFunc")`.

You can use funcref with maps and filters. Note that maps and filters will pass an index as the first argument and the iterated value as the second argument.

```
function! Breakfast(index, item)
  return "I am having " . a:item . " for breakfast"
endfunction

let breakfast_items = ["pancakes", "hash browns", "waffles"]
let first_meals = map(breakfast_items, function("Breakfast"))

for meal in first_meals
  echo meal
endfor
```

Lambda

A better way to use functions in maps and filters is to use lambda expression (sometimes known as unnamed function). For example:

```
let Plus = {x,y -> x + y}
echo Plus(1,2)
" returns 3

let Tasty = { -> 'tasty'}
echo Tasty()
" returns "tasty"
```

You can call a function from insisde a lambda expression:

```
function! Lunch(item)
  return "I am having " . a:item . " for lunch"
endfunction

let lunch_items = ["sushi", "ramen", "sashimi"]

let day_meals = map(lunch_items, {index, item -> Lunch(item)})

for meal in day_meals
  echo meal
endfor
```

If you don't want to call the function from inside lambda, you can refactor it:

```
let day_meals = map(lunch_items, {index, item -> "I am having " . item . " for lunch\
"})
```

Method Chaining

You can chain several Vimscript functions and lambda expressions sequentially with `->`. Keep in mind that `->` must be followed by a method name *without space.*

```
Source->Method1()->Method2()->...->MethodN()
```

To convert a float to a number using method chaining:

```
echo 3.14->float2nr()
" returns 3
```

Let's do a more complicated example. Suppose that you need to capitalize the first letter of each item on a list, then sort the list, then join the list to form a string.

```
function! Capitalizer(word)
  return substitute(a:word, "\^\.", "\\u&", "g")
endfunction

function! CapitalizeList(word_list)
  return map(a:word_list, {index, word -> Capitalizer(word)})
endfunction

let dinner_items = ["bruschetta", "antipasto", "calzone"]

echo dinner_items->CapitalizeList()->sort()->join(", ")
" returns "Antipasto, Bruschetta, Calzone"
```

With method chaining, the sequence is more easily read and understood. I can just glance at `dinner_items->CapitalizeList()->sort()->join(", ")` and know exactly what is going on.

Closure

When you define a variable inside a function, that variable exists within that function boundaries. This is called a lexical scope.

```
function! Lunch()
  let appetizer = "shrimp"

  function! SecondLunch()
    return appetizer
  endfunction

  return funcref("SecondLunch")
endfunction
```

`appetizer` is defined inside the `Lunch` function, which returns `SecondLunch` funcref. Notice that `SecondLunch` uses the `appetizer`, but in Vimscript, it doesn't have access to that variable. If you try to run `echo Lunch()()`, Vim will throw an undefined variable error.

To fix this issue, use the `closure` keyword. Let's refactor:

```
function! Lunch()
  let appetizer = "shrimp"

  function! SecondLunch() closure
    return appetizer
  endfunction

  return funcref("SecondLunch")
endfunction
```

Now if you run `echo Lunch()()`, Vim will return “shrimp”.

Learn Vimscript Functions The Smart Way

In this chapter, you learned the anatomy of Vim function. You learned how to use different special keywords `range`, `dict`, and `closure` to modify function behavior. You also learned how to use lambda and to chain multiple functions together. Functions are important tools for creating complex abstractions.

This concludes this Vim guide. However, your Vim journey doesn’t end here. In fact, it actually starts now. You should have sufficient knowledge to go on your own. You may even create your own plugins. Learning Vim is a lifelong pursuit, so never stop learning!

Happy Vimming, friends!