

Chapter 1: Installation

So you think you're ready to install Neovim? Actually, you still have one decision to make.

Neovim can run in a lot of different contexts (You can even run it inside VS Code)! By default it is a terminal program, but there are also tons of GUIs available. I have tried almost all of them, and, honestly, I don't think they have any inherent advantage over running Neovim directly in the terminal.

We will discuss hooking up NeoVim to the Neovide GUI much later in the book, but for starting out, I recommend running NeoVim in a terminal. A very good terminal, to be specific.

Choosing a Terminal

To get the best Vim editing experience, you want a GPU accelerated terminal. What's that mean? Basically that you will be using the chip designed to render photo-realistic video for rendering source code. Makes as much sense is using it for AI, right?

You will need to do your own research on the following options, so ask your favourite search engine or the AI Chat bot de jour to help you decide:

- [Kitty Terminal](#) is my personal preference. I find it well-documented, easy to configure and has all the features I need.
- [Alacritty](#) is probably the winner for raw speed, but configuration is awkward, and it is less featureful.
- [Wezterm](#) has some very nifty features, but I found the documentation to be lacking and had trouble getting some aspects to work.
- [Windows Terminal](#), if you are a Windows user, does claim to be GPU accelerated, though I found that Neovim was sometimes unresponsive in it.
- If you're already on the [Warp Terminal](#) train and you just can't live without it, Neovim *will* work inside it. I found the experience a little choppy, and I didn't enjoy the look and feel of Warp (or the fact that I needed to sign in to use it)

So install one or more of those until you find one that you like. You *can* use other terminal emulators if you want to. You probably won't even notice that the experience is inferior, but I can promise that if you later switch to a GPU-accelerated experience, you'll notice the improvement.

Setting Up Your Terminal Font

LazyVim and its plugins look beautiful in a terminal, and you would almost not believe that they are not a GUI application. To do this, they depend on special fonts that have a TON of glyphs for coding related things. Most notably, this gives you access to icons representing the type of file you have open, but also provide nice frames and window-like behaviour in the terminal.

To get the best LazyVim experience, you will need to install one of these special fonts and configure your terminal to use it. Indeed, you should really be using one of these fonts in your terminal even if you aren't a heavy Neovim user. A lot of modern terminal apps (there's a phrase, eh?) look better if you have them installed.

Visit [Nerd Fonts](#) for more information and to choose a font. I personally use the VictorMono Nerd Font because it has a unique typeface for the italic font that I like for code comment blocks.

You can choose from many of the most popular programming fonts. Downloading and installing them is very much operating system dependent, so I'll leave the Nerd Fonts website to explain it to you.

Install Neovim

Installing Neovim is generally one of the least problematic installs you will encounter as Neovim works pretty much anywhere software can be installed, and it only relies on standard system dependencies. The chief problem is that no matter which operating system you use, you are spoiled for choice!

You can visit the [Neovim home page](#) and click the "Install Now" button to get the latest instructions for your operating system of choice (or of necessity) from the Neovim developers.

Which Version should I install?

Neovim development happens at a super fast pace compared to their release cycle, so it is not uncommon for folks to run the latest nightly build. I have only rarely encountered bugs in builds cut from the master branch on Github, so it's generally safe. I usually run off the latest stable release when it comes out, and then when some new plugin update says "here's a cool feature if you use Neovim nightly," I'll install the latest build instead.

I suggest starting with the stable version of Neovim for now, which at time of writing is 0.9.5. The 0.10 release does have some cool features that LazyVim leverages, so if you start

to get as excited about this distro as I am, it will be a perfectly natural progression to switch to the pre-release.

Windows

I generally recommend using the Windows Subsystem for Linux (WSL) and doing all development in there. WSL is way outside the scope of this book, but it is well-documented by Microsoft and many online tutorials. Once you have chosen a WSL-compatible Linux distribution, set it up, and have it running in your chosen terminal, you can install Neovim using the Linux instructions below.

If you have a reason to—or preference for—developing on native Windows, the easiest thing to do is grab the MSI installer from the [Releases](#) section of the `neovim/neovim` repository on GitHub.

If you already use Winget, Chocolatey, or Scoop to manage packages on your Windows machine, there is a Neovim package in each of them.

Note that if you use Windows without WSL, you will need to install a C compiler in order to get treesitter support. This is not a trivial task. It is documented in the [nvim-treesitter/nvim-treesitter](#) GitHub repo, so I won't go into detail here.

MacOS

I recommend first installing Homebrew if you don't have it already by following the instructions at [brew.sh](#).

Once you have brew up and running, the command `brew install neovim` will install Neovim.

If you want to live on the edge, `brew install --HEAD neovim` will install the latest nightly version of Neovim, which is probably, but not guaranteed to be, stable.

I find the brew experience to be much kinder than other MacOS installation options for Neovim, so if you aren't already a Homebrew user, I strongly suggest exploring setting it up. There are other open source tools that you will want to install as we get deeper into the LazyVim journey, and brew will be the easiest way to get them all.

If you don't want to use Homebrew, things are a bit more annoying. The Neovim dev team doesn't maintain a MacOS installer, so you'll have to download a tarball and extract it, then link to the binary from somewhere on your system path. If you don't know what any of that means, honestly, use Homebrew, it's easier!

Linux

If Neovim isn't available using your distribution's default package manager, you have a very strange Linux distribution, indeed!

So just run `sudo pacman -S neovim`, `sudo apt install neovim`, `sudo dnf install neovim`, or the appropriate command for whichever more esoteric package manager you prefer.

If you want a nightly version, you may find the instructions on the [neovim/neovim GitHub Releases page](#), or will have to dig into your distro's documentation.

You will also need to install a C compiler in the unlikely event that your Linux distribution didn't come with one. For most distros, just install the `gcc` package and you should be good to go.

Try Neovim Raw (If You Dare)

Once you have Neovim installed, you can try it out by simply typing `nvim` (or `nvim <filename>` to open a specific file) into the terminal you installed a few sections ago. If it is installed correctly and on your path, you'll get an unappealing looking editor that looks like it was forked from something written in the 90s that had the express intent of looking like it was written in the 70s.

So, at least it's honest?

Unfortunately, you're now trapped. To save you the frantic "how do I exit vim" google search, the command to quit is `Escape` followed by the three characters `:q!` followed by `Enter`: `<Escape> <Colon> q <Exclamation> <Enter>`.

Seriously, "How do I exit vim" is one of the top three autocompletes on Google for "How do I exit...". Apparently only a Samsung TV plus and full screen mode on MacOS are harder to get out of!

TIP: If you want to, you can run the command `<Escape>:Tutor<Enter>` to open an interactive text file that you can read through and edit while learning the basics of Neovim. I do recommend doing this at some point, but now may not be the right time, as a lot of things that are "normal" in the vim tutor are different (better!) using LazyVim. The rest of this book does **not** assume you have gone through the tutor, but it also won't necessarily cover everything that is available there.

Install LazyVim

Now that you have Neovim up and running, let's get it configured to look like it was developed this century.

Installing LazyVim requires a bit of work with `git`. Since you are reading this book, I am assuming you are a) a software developer and therefore b) familiar with `git`. You can probably use whichever visual git tool you are familiar with. But... now that you have that fancy GPU accelerated terminal emulator, I say put it to good use.

The git commands to install LazyVim are more or less the same for the various operating systems, though paths and environment variables are different.

Start with a clean slate

First, remove or back up all existing Neovim state. This step is largely optional if you've never used Neovim before, but I recommend making sure the following directories have been removed or moved:

Clean up: Windows with Subsystem for Linux, MacOS, and Linux

```
rm -rf ~/.config/nvim
rm -rf ~/.local/share/nvim
rm -rf ~/.cache/nvim
```

If you aren't brand new to Neovim and have a config you want to keep in case you don't like this LazyVim experiment, move it someplace safe instead of removing it.

Clean Up: Windows without WSL

The location of the config and data folders is a little bit different, but the idea is the same as for the Unix systems. Just use Powershell commands instead of the Unix core-tools:

```
Move-Item $env:LOCALAPPDATA\nvim $env:LOCALAPPDATA\nvim.bak
Move-Item $env:LOCALAPPDATA\nvim-data $env:LOCALAPPDATA\nvim-data.bak
```

Install other recommended dependencies

I strongly recommend installing `lazygit`, `ripgrep` and `fd`, which are used by LazyVim to provide enhanced git, string searching, and file searching behaviours. Most package managers will have these available for trivial installation. You can find more specific installation instructions on their respective GitHub repositories under [jesseduffield/lazygit](https://github.com/jesseduffield/lazygit), [BurntSushi/ripgrep](https://github.com/BurntSushi/ripgrep) and [sharkdp/fd](https://github.com/sharkdp/fd) respectively.

Clone the starter template

You'll use a `git clone` command to download the starter template and copy it into the user config directory for Neovim, then remove the `.git` folder.

The starter is just that: a starter. So you won't ever need to pull changes from this repo. Instead, LazyVim will manage updating itself and all its plugins for you. The only reason the starter is a git repo is that it's easy for the LazyVim maintainers to maintain. From your point of view you're just downloading the current state of the repo and don't need to know about the past or future state.

git clone: Windows with Subsystem for Linux, MacOS, and Linux

On Unix systems, use these commands:

```
git clone https://github.com/LazyVim/starter ~/.config/nvim
rm -rf ~/.config/nvim/.git
```

git clone: Windows without WSL

On native Windows, use these commands:

```
git clone https://github.com/LazyVim/starter %env:LOCALAPPDATA\nvim
Remove-Item %env:LOCALAPPDATA\nvim\.git -Recurse -Force
```

The Dashboard

Ok, you have completed the most difficult section of this book and you're finally ready to start LazyVim! Use the same terminal command as before: `nvim`.

You'll see a flurry of activity as LazyVim sets everything up and downloads the plugins it thinks are essential. You may see it compile and install a bunch of treesitter grammars; if you see a message to "Show More" use `Shift+G` to skip to the end. Once everything is installed, you'll see a summary of the plugins that were installed inside a window managed by a plugin called `lazy.nvim`

The `lazy.nvim` plugin should not be confused with LazyVim itself, though both are maintained by the same person. `lazy.nvim` is strictly a plugin manager, whereas LazyVim is a collection of plugins and configurations that ship together. One of those plugins is `lazy.nvim`.

We'll be covering most of the plugins that ship with LazyVim later in this book, so for now, once you get to the `lazy.nvim` screen, you can press the `q`. The plugin will interpret this as `quit lazy.nvim` and the window will close.

Now you can see the LazyVim dashboard, which is the first thing you'll see every time you start LazyVim. It's a little more friendly than the out of the box Neovim experience:



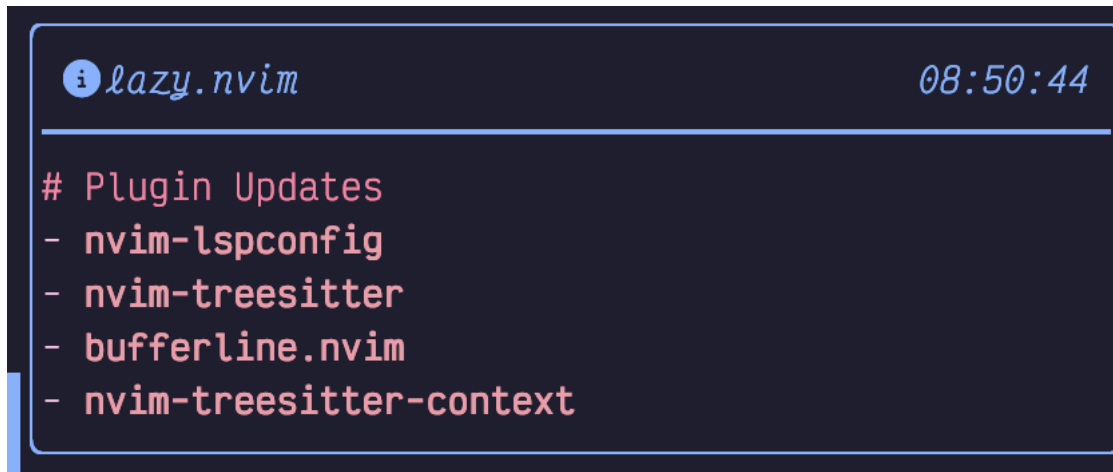
screenshot

As you can see, there are several commands that allow you to interact with the dashboard via a single keystroke. Most importantly, of course, is the `q` keystroke to quit!

Most of these options are self-explanatory, but we'll discuss a few of them more deeply in later chapters.

Lazy.nvim Plugin Manager

When you first open LazyVim, it checks for any plugins that are available to be updated, and gives you an overview in a message notification that will look something like this:

A screenshot of a message notification window from Lazy.nvim. The window has a dark background with light blue text. At the top left, there is an information icon (a lowercase 'i' in a circle) followed by the text 'lazy.nvim'. At the top right, the time '08:50:44' is displayed. Below a horizontal line, the text '# Plugin Updates' is shown in a light blue color. Underneath, a list of plugins is shown, each preceded by a hyphen: '- nvim-lspconfig', '- nvim-treesitter', '- bufferline.nvim', and '- nvim-treesitter-context'.

```
i lazy.nvim 08:50:44
# Plugin Updates
- nvim-lspconfig
- nvim-treesitter
- bufferline.nvim
- nvim-treesitter-context
```

screenshot

Because Neovim is pretty barebones by default, LazyVim ships with a ton of useful plugins ready to go. And there's a good chance they are out of date because plugin development in the Neovim world happens at a ridiculously fast pace.

In the old old days, plugin management was completely manual process. In the less old, but still old days, it was managed by a variety of plugins that did the job but felt like they were lacking something.

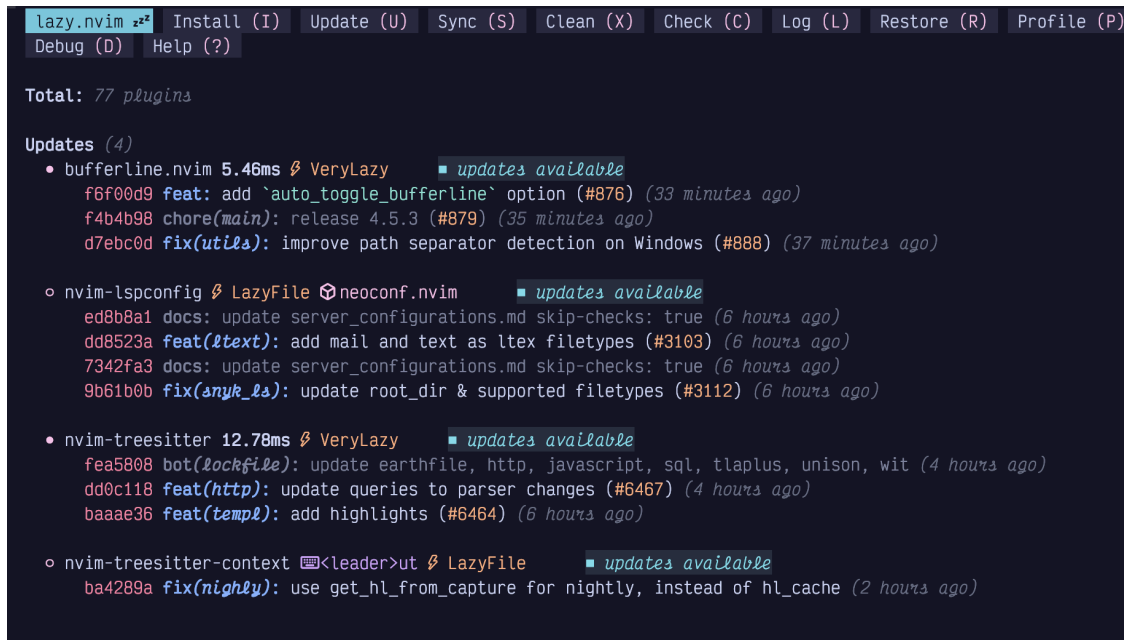
Then came the plugin manager called `lazy.nvim`, created by the same person that later created LazyVim.

`Lazy.nvim` has a ton of slick features, most notably loading plugins only when needed (hence the name "Lazy") so that your editor is lightning fast to start up. It also has a nice UI for managing plugins installation and updates.

You can access this UI from the dashboard simply by pressing the `1` key, which is labelled in the dashboard as `Lazy`. The label should probably be `Lazy PLugin Manager` to make it a bit more clear, but now you know what `Lazy` means so you won't forget.

If you are not actively displaying the dashboard, you can show the plugin manger at any time by entering space mode. We'll cover space mode in detail in the next chapter, but for now: First make sure you are in Normal mode by checking the lower left corner of the active window. If not, press `Esc` to enter Normal mode. Then press `Space` to enter space mode, followed by `l` to bring up the lazy.nvim plugin manager.

The Lazy plugin manager interface looks like this:



```
lazy.nvim zZ Install (I) Update (U) Sync (S) Clean (X) Check (C) Log (L) Restore (R) Profile (P)
Debug (D) Help (?)

Total: 77 plugins

Updates (4)
• bufferline.nvim 5.46ms ⚡ VeryLazy ■ updates available
  f6f00d9 feat: add `auto_toggle_bufferline` option (#876) (33 minutes ago)
  f4b4b98 chore(main): release 4.5.3 (#879) (35 minutes ago)
  d7ebc0d fix(utils): improve path separator detection on Windows (#888) (37 minutes ago)

○ nvim-lspconfig ⚡ LazyFile ⚡ neoconf.nvim ■ updates available
  ed8b8a1 docs: update server_configurations.md skip-checks: true (6 hours ago)
  dd8523a feat(ltext): add mail and text as ltex filetypes (#3103) (6 hours ago)
  7342fa3 docs: update server_configurations.md skip-checks: true (6 hours ago)
  9b61b0b fix(anyk_la): update root_dir & supported filetypes (#3112) (6 hours ago)

• nvim-treesitter 12.78ms ⚡ VeryLazy ■ updates available
  fea5808 bot(lockfile): update earthfile, http, javascript, sql, tlaplus, unison, wit (4 hours ago)
  dd0c118 feat(http): update queries to parser changes (#6467) (4 hours ago)
  baaae36 feat(templ): add highlights (#6464) (6 hours ago)

○ nvim-treesitter-context 🗨️<leader>ut ⚡ LazyFile ■ updates available
  ba4289a fix(nightly): use get_hl_from_capture for nightly, instead of hl_cache (2 hours ago)
```

screenshot

The window that has popped up is called a floating window. You'll see these in a few different situations, usually when there is interactive data that you need to work with (like a web modal). This particular floating window comes with it's own set of keybindings. The keybindings are listed across the top, and pay attention to the fact that all of them are capitalized, so you need to use the `Shift` key when invoking them.

Realistically, the only keybinding I use on a regular basis is `Shift-S`, for `Sync`. This is the equivalent of running `install`, `clean`, and `update` in one single action, so it has the effect of guaranteeing that the versions of plugins that are actually installed is exactly consistent with the ones specified in the LazyVim configuration.

So when the "Plugin updates available" notification pops up, just press `Space-l` and then wait for the sync to complete. Then press `q` to close the Lazy.nvim plugin mode and floating window and return to what you were doing.

A Note on Managing Dot Files

If you work on multiple different computers, you'll quickly find that you don't want to set up your LazyVim configuration separately on all of them. LazyVim does not have the equivalent of VSCode's "settings sync", though such plugins exist.

An alternative I recommend instead is to store your config files in a git repository. You'll probably find there a few other files you want to keep in there such as your `.gitconfig` and `.zshrc` / `.bashrc` / `.config/fish/config.fish`. If you use GitHub Codespaces, you may already manage some dot files with git.

If not, my personal recommendation is to follow the advice in the excellent blog article [Dotfiles: Best way to store in a bare git repository](#) from the Atlassian blog.

Before distributions like LazyVim came along, it was very common for people to store their vim configuration in a public repository, and borrow ideas from each other. This practice is not quite dead, and you can find my own dot files on GitHub in the [dusty-phillips/dotfiles](#) repository.

Summary

In this chapter, we briefly discussed the history of Vim, Neovim, and LazyVim, and why they are still relevant today. Then we covered the importance of GPU accelerated terminals and Nerd Fonts.

We figured out how to install Neovim and its dependencies under whichever operating system(s) you use, and finally, installed LazyVim from its starter template.

In the next chapter, we'll discuss Vim's core feature: Modal Editing, and dig into the many things you can do with your keyboard in LazyVim.

Chapter 2: What is Modal Editing, Anyway?

As you may have guessed from the letters on the dashboard, LazyVim is very keyboard-centric. As many actions as possible can be performed without moving your hands between mouse and keyboard. That's not to say that it's impossible to use the mouse. You can click anywhere in the editor, interact with buttons and modals when they pop up, use the scroll wheel or gestures to scroll, and resize editor panes by dragging their borders, for example. But you can also do all of these things using the keyboard, and usually more efficiently.

More importantly, you can do most things by holding at most two keys, and usually just one. You will only rarely have to contort your hands into painful (and dangerous) positions to Control + Shift + Alt + <some key>.

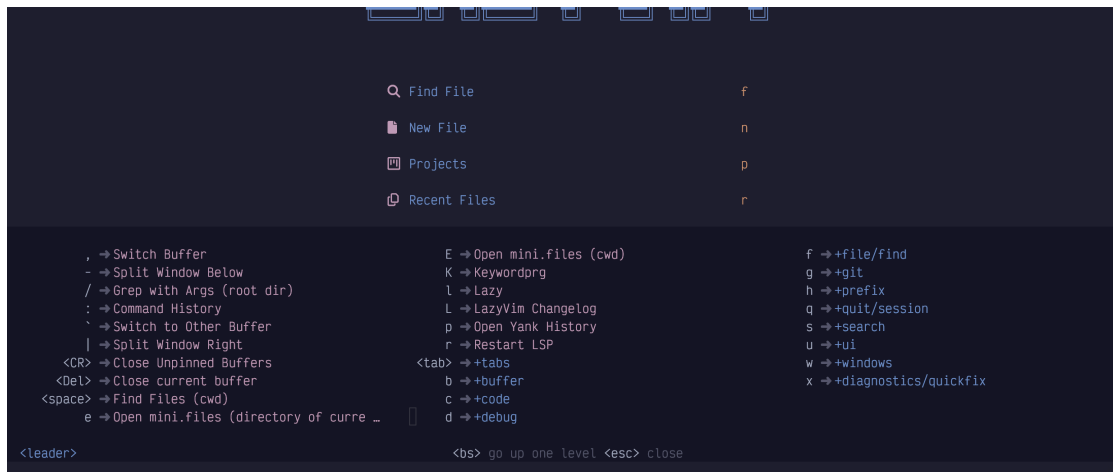
How does Vim do this? Modal editing.

Introduction to Modal Editing

“Modes” in LazyVim simply mean that different keystrokes mean different things depending on which mode is currently active. For example, when you start the editor up, you are in a “Dashboard Mode”, and the most common interpretation of keystrokes in that mode are listed right there on the dashboard. This discoverability of keybindings in a given mode is a common theme in LazyVim, and a huge improvement over the opaque default behaviour of Neovim itself.

To see what I mean, press the spacebar to enter “space mode”. Space mode is a LazyVim concept; it does not exist in a raw Neovim installation (though you can install various plugins to recreate the effect if you want Neovim without LazyVim).

Entering space mode pops up a menu along the bottom of your screen. If you have the dashboard open, it will look something like this (my menu contains some customizations, so yours won't be identical):



screenshot

That's a big menu. The important thing to focus on right now is the `f` key, which we will use to understand modal editing.

If you are in dashboard mode and press the `f` key, you will open the `Find file` dialog using a plugin we'll discuss later called Telescope. However, now that you are in space mode, if you press the `f` key, it will open the `file/find` space mode submenu.

This is the crux of what modal editing means: The behaviour of a given key depends on the current mode. As indicated by the line at the bottom of the space mode menu, you can press the `Escape` key to exit space mode and return to the dashboard. Go ahead and do that.

Now you're back in dashboard mode, and you can press the `n` key to create a new, empty buffer.

Pay close attention to the lower left corner of that buffer, where you'll see the word `INSERT` in green:



screenshot

Remember how I said Space mode is a LazyVim concept? Insert mode is a Vi concept, that the successors Vim, then Neovim, and now LazyVim have all inherited. In Insert mode, the vast majority of keystrokes do what you would expect in any editor: they insert text. So you can touch type as with any other editor!

You can access *some* keyboard shortcuts in insert mode using `Control` and `Alt` keys. For example, you can hit `Control-r` to enter the "Registers" mini-mode, which pops up a list of "registers" you can paste from. We'll cover registers in detail later. For now, it is enough to know that `Control-r` followed by the plus key (i.e. `Shift-=`) will paste text from the clipboard in insert mode.

However, you will much more often change to *Normal* mode to perform any non-text-entry operations, including pasting text.

To get into Normal mode from Insert mode, hit the `Escape` key. The cursor will change from a bar to a block and the indicator in the lower left corner will change to a blue `NORMAL`:



screenshot

In Normal mode, pressing various keyboard characters will not insert text like it does in Insert mode. For example, pressing `p`, rather than inserting a literal `p` character into the document, will instead paste from the system clipboard.

Vim and Neovim aren't very discoverable, but they ARE extremely memorable. As often as possible, the keyboard shortcuts to perform an action start with a letter that makes sense for the action being performed. You might think `p` stands for "Paste", but in fact the concept has been around for longer than the clipboard mnemonic. You are welcome to think of it as "paste" if that's easier for you, but in vim parlance, it actually stands for "put", and we'll use that word in different contexts throughout the book.

For some contrast, the `Control-r` key that pops up the list of registers in Insert mode does **not** pop up a list of registers in Normal mode. Instead, `Control-r` means "Redo" (aka undo an undo). In order to enter the Registers mini-mode from Normal mode, you would press the `"` (Shift-apostrophe) key instead.

If that sounds confusing, don't worry. Your brain and muscle memory will adapt more quickly than you expect and you'll always understand that behaviours in normal mode are not the same as in insert mode.

To be honest, I hardly ever use non-text-entry commands in Insert mode. I find it is easier to switch back to Normal mode and then perform the command from normal mode. It doesn't usually take a higher number of keystrokes to do so and I don't have to hold down multiple keys at once.

As I mentioned, the universal key to exit Insert mode and return to Normal mode is `Escape`. And that brings us to an important point: You will be using this key a lot, but moving your hands from the home row to the `Escape` key in the upper left corner and back again is somewhat inefficient.

There are a few common workarounds to this situation:

- If you have a customizable keyboard you can put the escape key in a more accessible location. This is what I do. I have a Kinesis Advantage 360, and I remapped the keys so that escape is in the "thumb key" section of this admittedly bizarre keyboard. It's as easy to hit as enter, space, and backspace, other keys that I use very frequently.
- Your operating system is probably also capable of remapping keys. A lot of users replace the largely useless `Capslock` with the `Escape` key. If you ever go back to the keyboard chaining editors descended from Emacs, including VSCode: For these editors it can be more comfortable to remap `Capslock` to the commonly-held `Control` key, especially on laptop keyboards).

- Neovim itself is also able to remap keys. We'll discuss how to do this in LazyVim later. One common pattern is to map a series of uncommon keystrokes that you wouldn't likely type together when inserting text to the escape key. So you can set it up to map something like `jk`, `jj` or `;` in Insert mode to switch to normal mode. I've tried this and don't care for it as it introduces a timing thing when you hit the first character and Neovim is waiting to see if you're going to type a command or let text insertion continue, but you might like it.
- The `Control-C` keyboard combination also works to exit normal mode, with no remapping required. I don't like this because it's two keystrokes and on my Dvorak keyboard, `Control-C` is harder to hit than on a qwerty keyboard where `C` is on the bottom row near the `Control` key.

Don't worry about actually changing it for now; just start getting used to using `Escape` where it is and see if you find it annoying. I'll mention it again when we get to the remapping section and you can decide then if you want to change it.

Once you're in Normal mode, you'll obviously want to get back to Insert mode to enter text at some point! There are several different ways to do this that we'll discuss later. As a taste, here are a couple of the most common ones:

The `i` key always inserts text *before* the current cursor position. This means that you could (very clumsily) move your cursor left by pressing `i <Escape> i <Escape>` repeatedly. When you press `i`, you insert text before the current position, and then `escape` takes you out of Insert mode at that new "before" position.

Commonly, you want to enter insert mode *after* the current cursor position. To do that, use the `a` key instead (mnemonic: `i` = Insert Before, `a` = Insert **A**fter).

You'll find that you need to alternate between these a lot as you are navigating a document because the various navigation commands we'll cover later will often put you just before or just after the position you need to insert at. So it's important to remember both of them.

Two other very common operations are to insert at the very beginning or the very end of the current line. You *could* use navigation commands to move to the start or end and then use `i` and `a`, but it's easier to use the commands `Shift-I` and `Shift-A` instead (The difference is that they are capitalized, so you need the Shift key with them).

A note on Keybinding Mnemonics

It is very common for related keybindings like these to be assigned to the lowercase and uppercase versions of the same key. You will often find that the lower case version means "do something" and the uppercase version means either do the same thing only BIGGER or do the opposite thing, depending on the situation. In this case, `i` and `a`

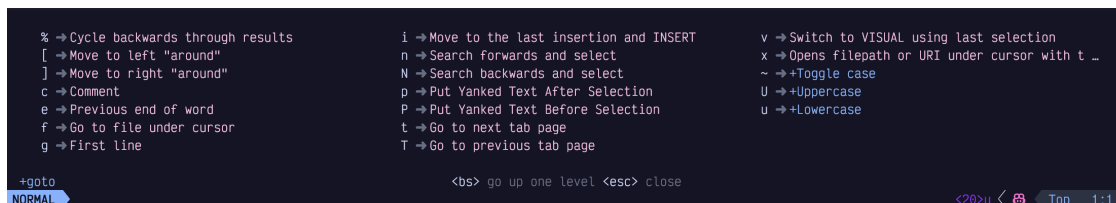
mean “insert one character before or after the cursor” and I and A are “insert before or after the cursor, only BIGGER (i.e. at the beginning or end of the line)”.

To illustrate the “do the opposite thing” situation, consider the o and Shift-O keys, which are two new ways to get into insert mode.

The o key is used to enter insert mode on a new line below the current one. I’ve heard the mnemonic as “**O**pen a new line above/below” to help you remember the otherwise not terribly memorable o command. And in the classic “do the opposite thing” scenario, Shift-O means “create a new line *above* the current one and enter insert mode on it”.

Let’s discuss one final *very* useful command takes two keystrokes, one after the other: gi. That is a single press and release of g followed by i.

This effectively means “Go to the last place you entered insert mode, and enter insert mode again”. In this case, the g key is actually switching to a new mini-mode I call “Go To” mode, though not all the commands accessible from it are strictly related to going places. You can see the entire list of commands available in “Go To” mode by pressing the g key in normal mode and waiting for the menu to pop up at the bottom of the window:



```
% → Cycle backwards through results
[ → Move to left "around"
] → Move to right "around"
c → Comment
e → Previous end of word
f → Go to file under cursor
g → First line

i → Move to the last insertion and INSERT
n → Search forwards and select
N → Search backwards and select
p → Put Yanked Text After Selection
P → Put Yanked Text Before Selection
t → Go to next tab page
T → Go to previous tab page

v → Switch to VISUAL using last selection
x → Opens filepath or URI under cursor with t ...
~ → +Toggle case
U → +Uppercase
u → +Lowercase

+gto
NORMAL
```

screenshot

We’ll cover most of them later, but notice that the i key is in there labelled *Move to the last insertion and INSERT*. So if you forget how to go to the last insertion point, you can enter Go To mode and scan the menu to find the i again.

Try all of those commands (a, i, o, A, I, O, and gi) repeatedly, entering some text and pressing Escape to return to Normal mode. Then try it again. Move your cursor around the text using the mouse (we’ll get to keyboard navigation soon, I promise), and try using the commands again to see how they behave in new locations.

Get *really* comfortable with switching between Normal and Insert mode. You might think you’ll spend most of your time in Insert mode, but the truth is code is edited far more often than it is written afresh, and you’ll be alternating between them constantly.

Visual Mode

The other major mode that LazyVim inherits from its ancestors is “Visual” mode. Visual mode is used to select text. In general, you can enter Visual mode and then use many of the same navigation keys you would use in Normal mode to move your cursor around. Since we haven’t covered those navigation keystrokes yet, I’m going to defer a detailed discussion of Visual Mode until we have the necessary foundation.

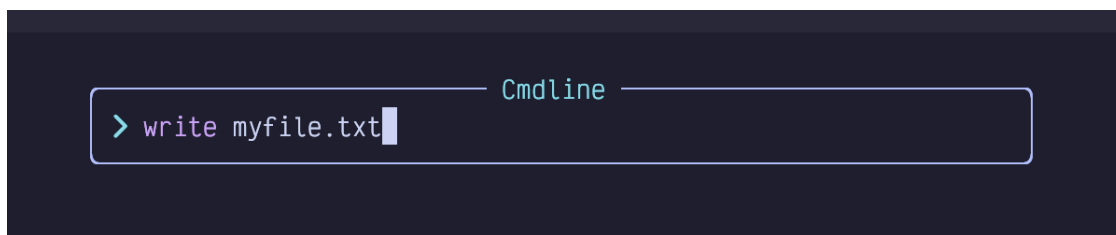
Command Mode

Command mode is different from some of the other modes we’ve seen, which were mostly submenus or editor-level major modes. You can get into command mode from Normal mode by using the `:` (i.e. `Shift-<semicolon>`) command. In LazyVim, this will pop up a little widget where you can type what is known as an “Ex Command.” This name comes from `vi`’s predecessor, `ex`, which hasn’t really been used (other than as part of `vim`) in decades.

Essentially, you can enter a wide variety of commands into this widget and expect certain behaviours to happen as a result. It is actually more similar to the VS Code command pallet than anything else, though it is a quite different beast.

You already know one ex command from the previous chapter! Remember `<Escape><Colon>q!<Enter>` the command to exit the editor? You now know that the `Escape` is to enter Normal mode from whatever mode you are in. The colon is used to switch to Command mode, and the `q` is short for quit (You could type the full word `quit` if you didn’t feel the need to conserve keystrokes). The exclamation point says “without saving” and the `Enter` means “submit the ex command”.

As another example, let’s consider the `write` ex command. Type `:` followed by `write myfile.txt` like this:

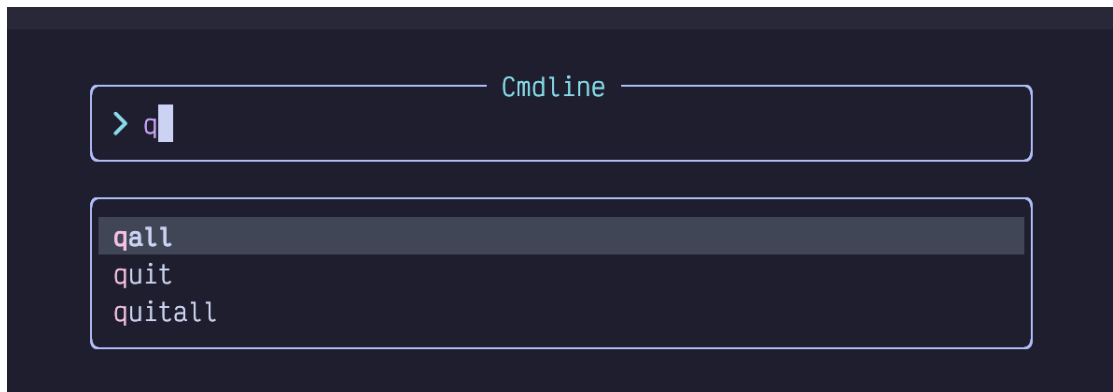


screenshot

Press `Enter` to confirm and execute the command.

Note: Most commands can be shortened to their shortest unique common prefix. You can type `:w myfile.txt` instead of `:write myfile.txt`. The most popular commands even have special combined commands, so `:wq` will save and exit, although you'll probably prefer `:x` as it's even shorter.

Command mode is kind of weird because it's kind of like an insert mode in the sense that you can type text into it, and some of the keybindings that work in insert mode work also in command mode (including `Control-r` to paste from a register). But other keybindings work differently in command mode. The most important one is the `Tab` key, which will do a sort of "tab completion" on the command. For example, `:q<Tab>` pops up a menu like this:



screenshot

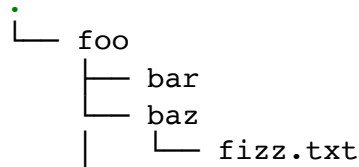
This damn completion menu is surprisingly unintuitive to navigate. You're probably going to want to bookmark this section or take some notes or something until you get used to it!

First, if you want to select a different entry in the menu, you would surely think you can use the arrow keys. Which you can, but it's a mind-mess because you need to use `Left` and `Right` to move the cursor `Up` and `Down`. I know! WTF, right?

This is mostly because the menu looks different in LazyVim than it did in normal Neovim, but the keys haven't been remapped. So instead, I suggest using `Tab` and `Shift-Tab` to select different entries from the menu. It's easier to remember and much easier on the muscle memory: `Tab` once to show the menu, `tab` again to cycle through the menu.

Second, there is some nuance around *confirming* one of those menu entries. In the above example, you can just press `Enter` to confirm the selection **and execute it**. However, there are often cases where you want to confirm the selection and then continue editing the command. An excellent example is the `:e` or `:edit` command.

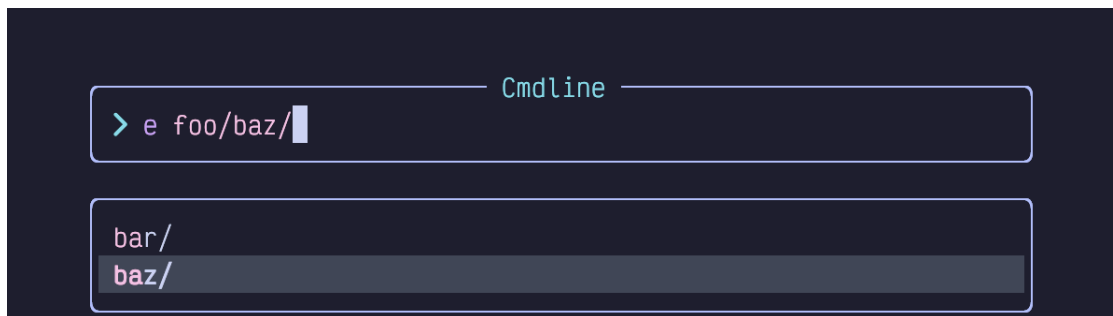
This command is used to open a file on your filesystem, but you have to type the entire path to the file. For example, if you have the following directory structure:



...and you have Neovim open, you would have to type the following to open the `baz.txt` file:

```
:e foo/baz/fizz.txt
```

That's a lot of typing if you need to get to deeply nested directories. Luckily, you *can* use tab completion for this. You can type `:e f<tab>b<tab><tab><tab>` to get `foo/baz`, but at this point the menu is still open:



screenshot

If you press `Enter` now, it's going to open the `baz` folder instead of just confirming the selection, which is not what you want. And if you press `Tab` again it will cycle through the menu some more. What you want to do at this point is press the `Control-y` (`y` for "yes") key combination. This will confirm the `baz` selection and close the menu. Now you can press `tab` again to complete the `fizz.txt` portion of the command.

It **is** possible to remap these keys to be more like other software, and I honestly think this is one thing LazyVim should do by default. I personally, have not remapped them as I have plenty of vim muscle memory using these keystrokes, but I will show you how to do it later when we cover keymapping.

You probably won't spend a lot of time in command mode. There are easier ways to open files in LazyVim, for example, as well as to quit the editor. And if you need to do something

more complex with command history, there is a special window you can use to edit commands with Insert and Normal mode that we will cover later.

For now, remember `<Tab>` and `Control-y` and you'll be able to navigate the Command menu when you need to. There are other keybindings you can use to edit commands, but unless you find yourself annoyed by certain repeated tasks, I wouldn't worry about them.

The most important command, by the way, is `:help`. Vim was created before folks had ready access to the Internet, so it has a tradition of shipping all of its documentation with the editor. Plugins tend to follow this tradition. So for example, if you can't remember the keyboard shortcut to put text, try `:help put`. Or, if you want to know what the `Control-R` keyboard shortcut does, try `:help Control-R`. Of course, the Vim help documents have been indexed by your favourite search engines and AI chat bots, so you can go all new-school and ask them if you prefer.

Summary

In this chapter, we became comfortable with the concept of modal editing and the most important LazyVim modes. There are other mini-modes and one major mode that will come up as we progress through this book, but becoming comfortable with Normal, Insert, and Command mode (and how to switch between them) will take you a long way on your LazyVim journey.

In the next chapter, we'll learn a whole bunch of different ways to move the cursor around inside a document.

Chapter 3: Getting Around

Software developers spend far more time editing code than we do writing it. We're always debugging, adding features, and refactoring.

Indeed, the most common thing I ever do is add a `print/println/console.log` at some specific line in the codebase.

If you are coming from the more common word processing or text editing ecosystems, navigating code is the thing that is most different in vim's modal paradigm. Even if you're used to vim, some of the plugins LazyVim ships by default suggest different methods of code navigation from the old vim standbys.

In VSCode, often the quickest way to get from one point in the code to another is to use the mouse. For minor movements, the arrow keys work well, and they can be combined with `Control`, `Alt`, or `Cmd/Win` to move in larger increments such as by words, paragraphs, or

to the beginning or end of the line. There are numerous other keyboard shortcuts to make getting around easier, and the Language Server support allows for easy semantic code navigation such as “Go to Definition” and “Go to Symbol”.

Vim also supports mouse navigation, but you’ll likely reach for it less often once you train up on the navigation keymappings. Vim has keybindings for the same Language Server Protocol features that VSCode has, and they are often more accessible. The big difference with Vim is the entire keyboard’s worth of navigation commands that are opened up to you when your editor is in Normal mode.

Seeking Text

LazyVim ships with a plugin called `flash.nvim`, which is maintained by the creator of LazyVim and integrates very nicely with it.

This plugin provides a code navigation mode that has been available in various vim plugins (starting with one called EasyMotion) for many years, and has historically been quite controversial. A lot of long-time vim users think it breaks the vim paradigm. I won’t go into the details as to why, but I will acknowledge that this was true in older iterations of the paradigm and is much less true in modern versions such as `flash.nvim`.

If you can see the code you want to navigate to (i.e. because the file is currently open and the code is scrolled into view), `flash.nvim` is almost always the fastest way to move your cursor there. It admittedly takes at least three keystrokes, but those three keystrokes require no mental math or incrementally “moving closer” to the target until you get there, which are two of the less efficient problems that come up with certain other vim navigation techniques (as well as non-modal editing).

To invoke `flash`, press the `s` key in Normal mode. My mnemonic for `s` is “s stands for seek”, although I’ve also heard it referred to as “sneak” or “search” mode. Searching in LazyVim is a different behaviour (it doesn’t care if the text is currently visible or not), and “sneaking” sounds a little too dishonest, so I use “Seek”.

The first thing to notice when you press `s` is that the text fades to a uniform colour and there’s a little lightning symbol in the Mode indicator indicating that Flash mode is active:

```
    {:#if $selectedTag}
      <TagAvatar size="xs" tag={
    {:#else}
      <UserSolid /> Selected Tag
    {:/if}
  {:#else if $outlineFilter ≡ '
    <CheckCircleOutline /> Outsta
```

screenshot

Since you know where you want the cursor to be, your eyes are probably looking right at it, and you know exactly what character is at that location. So after entering seek mode, simply type the character you want to jump to.

For example, in the following screenshot, I want to fix the (intentional) typo in the heading of this section, changing `Test` to `Text`.

```
more accessible. The big difference with Vim is the entire keyboard's worth of navigation commands that are opened up to you when your editor is in Normal mode.
```

```
## Seeking Test
```

```
LazyVim ships with a plugin called flash.nvim, which is maintained by the creator of LazyVim and integrates very nicely with it.
```

```
This plugin provides a code navigation mode that has been available in various vim plugins (starting with one called EasyMotion) for many years, and has historically been quite controversial. A lot of long-time vim users think it breaks the vim paradigm. I won't go into the details as to why, but I will acknowledge that this was true in older iterations of the paradigm and is much less true in modern versions of flash.nvim.
```

```
If you can see the code you want to navigate to (i.e. because the file is currently open and the code is scrolled into view), flash.nvim is almost always the fastest way to move your cursor there. It admittedly takes at least three keystrokes but those three keystrokes require no mental math or incrementally "moving closer" to the target until you get there, which are two of the less efficient problems that come up with certain other vim navigation techniques (as well as non-modal editing).
```



screenshot

I have hit `ss`, and every single `s` in the screenshot has turned blue, including capitals. There is an `s` character beside the flash icon in the status bar indicating that I have sought an `s`.

In addition, *beside* (to the right) of all the `s` characters nearest to the cursor (which is in the bottom paragraph) have a green label beside them. If I wanted to jump to any of those `s` characters, I would just have to type that label and boom, I'd be there.

However, the character I want to hit is too far away to have a unique label, as there are a lot of `s` characters in my text. No matter! I just have to type the character to the right of the target `s` character, which is a `t`. Now my screen looks like this:

```
more accessible. The big difference with Vim is the entire keyboard's worth of navigation commands that are opened up to you when your editor is in Normal mode.
```

```
## Seeking Testq
```

```
LazyVim ships with a plugin called flash.nvim, which is maintained by the creator of LazyVim and integrates very nicely with it.
```

```
This plugin provides a code navigation mode that has been available in various vim plugins (starting with one called EasyMotion) for many years, and has historically been quite controversial. A lot of long-time vim users think it breaks the vim paradigm. I won't go into the details as to why, but I will acknowledge that this was true in older iterations of the paradigm and is much less true in modern versions such as flash.nvim.
```

```
If you can see the code you want to navigate to (i.e. because the file is currently open and the code is scrolled into view), flash.nvim is almost always the fastest way to move your cursor there. It admittedly takes at least three keystrokes, but those three keystrokes require no mental math or incrementally "moving closer" to the target until you get there, which are two of the less efficient problems that come up with certain other vim navigation techniques (as well as non-modal editing).
```

```
⚡ st
```

screenshot

Now, all instances of `st` in the file are highlighted in blue, and since there aren't as many `st` as `s`, all of those instances have a label beside them. The text I want to move to is labelled with a `y`, so I press `y` and my cursor is moved to the `s` character I wanted to change. Now I can type `rx` to replace the `s` with an `x` (we'll discuss *editing* code in a later chapter, but now you've had a taste of it).

If you have multiple files open in splits (which we'll also discuss in detail later), seek mode can be used to move your cursor *anywhere* on the screen, not just in the currently active file.

Seek mode does have drawbacks however, at least the way `flash.nvim` implements it. There are some characters you can't move to directly because you run out of text to search for before a labelled match is in that location. For me this happens most often when I want to edit the end of a line. If I type `sn` because I want to edit a line that has `n` as the last character, but there are a bunch of `n` characters closer to my cursor than the one I want to move to, flash may not label the `n` I want to move to, and it won't accept a carriage return as a "next character" input.

For this reason, I don't seek near ends of lines. Instead, I'll seek to a word somewhere in the middle of the same line and then use `A` which, as you may recall, will put me in insert mode at the end of the line. Alternatively, if I don't want to enter insert mode, I will use the `$` symbol (`Shift+4`), which is the normal mode command for "Move cursor to end of current line".

Scrolling the screen

Seek mode only works if the text you want to jump to is visible on the screen. You can't label something you can't see! Often, this means you want to use search or one of the larger or more specific motions discussed later, but there are also a few keybindings you can use to scroll the screen so you can see your target and jump to it.

These keybindings are a little unusual by Vim standards because they mostly involve using the control key. How anti-modal! In my experience, these keybindings don't actually get a ton of use. Indeed I've forgotten some of them and had to look them up to write this chapter.

The scrolling keys I use the most are definitely `Control-d` and `Control-u`, where the mnemonic is **d**own and **u**p. They scroll the window by half a screen's worth of text. The cursor stays in the same spot relative to the **window**, which means that it is moved up or down by half a screen's worth of text relative to the **document**.

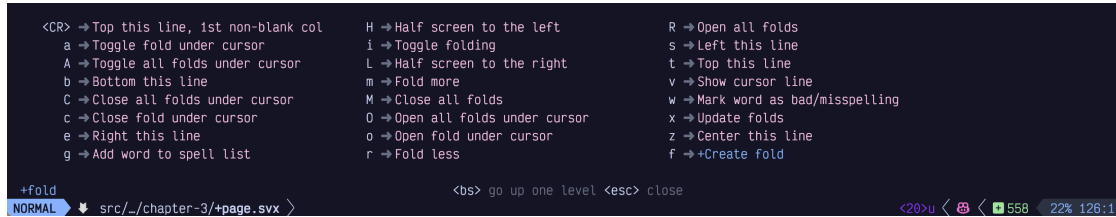
If you need to move even further, you can use the `Control-f` and `Control-b` keybindings, which move by a full page of text. I don't like these ones because I never quite know where the cursor is going to end up and I become disoriented. But it can be handy if you need to scroll something into view quickly to use Seek mode on it. Unlike `Control-d` and `Control-u`, `Control-f` and `Control-b` can be prefixed with a count, so you can type `5<Control-f>` if you need to scroll ahead by 5 pages.

I have no idea why the keys `Control-y` and `Control-e` were chosen to scroll the window by one line at a time. I never use them. These keybindings accept a count, so if you can remember them, they are useful for subtle repositioning of the text. The main advantage of these keybindings is that they don't move the cursor unless it would scroll off the screen, so if you are working on a line and need more visibility but don't want to move the cursor, you could use `Control-y` and `Control-e` to do it.

The reason I don't use these keys (other than lack of a decent mnemonic) is that I prefer to do relative cursor positioning using `z` mode.

Z Mode

The z menu is kind of an eclectic mix of cursor positioning, code folding, and random sub-menus, as you can see by pressing the z key while in normal mode:



```
<CR> → Top this line, 1st non-blank col      H → Half screen to the left      R → Open all folds
a → Toggle fold under cursor                i → Toggle folding                S → Left this line
A → Toggle all folds under cursor           L → Half screen to the right     t → Top this line
b → Bottom this line                        m → Fold more                     v → Show cursor line
C → Close all folds under cursor           M → Close all folds              W → Mark word as bad/misspelling
c → Close fold under cursor                D → Open all folds under cursor  X → Update folds
e → Right this line                        o → Open fold under cursor       Z → Center this line
g → Add word to spell list                 r → Fold less                     F → +Create fold

+fold                                     <bs> go up one level <esc> close
NORMAL  src/_/chapter-3/+page.svx  <20>u  558  22% 126:1
```

screenshot

If that looks like a big menu, you don't know the half of it! There are a ton of other z-mode keybindings that are obscure enough to not deserve mention in the menu! I'll cover the three most useful scrolling related ones here and we'll discuss others later.

The relative cursor keybindings I use exclusively are `zt`, `zb`, and `zz`. These move the line that the cursor is currently on to the top, bottom, or middle of the screen, respectively. When moving to the top or bottom it will leave a few lines of context above or below the cursor.

There are others that will also move the cursor to the first column of the window, but instead of memorizing those shortcuts, I recommend using `zt0`, `zb0`, and `zz0` instead. As we'll discuss later, the `0` command just means "Go to the start of the line" (You can also use `home` if your keyboard has a home key, but `0` is easier to hit on many keyboards).

You can find other scrolling keybindings in the NeoVim documentation by typing `:help scrolling`, but the ones I just mentioned will probably more than cover your needs as you learn far more nuanced methods of navigating code.

The first rule of Vim

So there is a holy rule in Vim that I constly break for valid reasons. Unless you are the very strange combination of weird that I am, you probably should not break it:

Never use the arrow keys to move the cursor.

The background behind this rule is that it takes a tenth of a second or so to move your hand to the arrow keys on most keyboards, and another tenth of a second to move it back to the home row. I'm not convinced these tenths of a second add up to an appreciable amount of time, even considering the millions of characters I have typed in my lifetime (Yes, millions. I did the math once).

But I do think the arrow keys on most keyboards can do nasty things to the long term health of your hands, and honestly, the more you get used to the alternative vim keybindings, the more you'll prefer to use them.

The vim keybindings for arrow keys seem rather unintuitive when you first look at them: h, j, k, and l. These map to the directions, left, down, up, and right. If it seems weird that l means "right" instead of left, or you're wondering why they skipped i since that appears to be an alphabetic sequence, look at your keyboard.

If you are an English-speaking person with a standard Qwerty keyboard, the letters h, j, k, and l are on the home row under your right hand, in that order, and are therefore the easiest keys to hit on the entire keyboard.

Open a largish file in Neovim (you can use `:e path/to/filename`) and experiment with moving the cursor left, right, up and down using the home row keys. While you do that, I'll tell you why I don't use them because I'm triply abnormal.

First, I'm left handed, so the right hand home row is slightly less accessible feeling. Second, I've been a Dvorak user for two decades. The j, k, and l keys are not on my home row. Third, I use a Kinesis Advantage 360 keyboard, which, among other bizarre layout features, places the arrow keys within reach of my fingers so I don't have to move my hand to hit them.

By a strange twist of fate, these weirdnesses kind of cancel each other out. The j and k keys happen to be directly above the left and right arrow keys under my dominant left hand. So that's what I use for navigation: Left Right, j k. If you are less weird than me, you should probably use the right-hand home row keys the way vim was designed.

Vim, Neovim, and LazyVim are all *really* good at reusing motions, so you will find that h, j, k, and l are used for a lot of different navigation sequences as you progress through this book. Take enough time to really get used to them. But recognize that if you ever have to push these keys more than twice in succession to move the cursor, you're wasting keystrokes.

Counting

The vast majority of commands in Vim can be prefixed with a *count* to repeat the motion multiple times. The count is typically entered as a sequence of digits before the command you want to repeat.

So, for example, to move the cursor up 15 lines, you would enter normal mode and hit the keys 15k. To move it five characters to the right, use 5l.

This is why LazyVim has such weird line numbering by default. Consider the following screenshot:

```
6 because I want to edit a line that has n as the last character, but there are
5 a bunch of n characters closer to my cursor than the one I want to move to,
4 flash may not label the n I want to move to, and it won't accept a carriage
3 return as a "next character" input.
2
1 For this reason, I don't seek near ends of lines. Instead, I'll seek to a word
126 somewhere in the middle of the same line and then use `A` which, as you may
1 recall, will put me in insert mode at the end of the line. Alternatively, if I
2 don't want to enter insert mode, I will use the $ symbol (Shift+4), which
3 is the normal mode command for "Move cursor to end of current line".
4
5 ## Scrolling the screen
```

screenshot

My cursor in this screenshot is on line 126, which is highlighted in the left gutter. It's also visible in the lower right corner of my window, though I cropped it out in this screenshot. But directly above line 126 we see the line number 1, and directly below it we also see the line number 1.

Let's say I want to move my cursor to the `Scrolling the screen` heading.

This line has the number 5 beside it, so I don't have to count lines or do any mental arithmetic to figure out the count to use to move my cursor. I just type `5j` and my cursor moves to the desired line.

Now that you know what they are for, I suggest leaving relative numbers on until you get used to them. If you find them distracting or just don't use them, you can change to normal line numbers by editing your LazyVim configuration. Open the file `~/.config/nvim/lua/config/options.lua`, which should have been created for you by LazyVim but currently won't have anything in it other than a comment describing what it is for.

Tip: You can use the space mode command `<Space>fc` to find files in the LazyVim configuration directory. This will pop up the Telescope file picker that we'll discuss in detail in the next chapter. Type `options` and press `<Enter>` to open the file.

To disable relative file numbers, add this line to the file and save it:

```
vim.opt.relativenumber = false
```

Then reopen Neovim, and you should see the absolute value of line numbers in the left column.

Personally, I find line numbers to not be very useful and I don't like wasting valuable screen width on displaying those characters. As has become a running theme, I recognize that I am somewhat odd! But if you also want to disable line-numbers altogether, you'll need another line in `options.lua`:

```
vim.opt.number = false
vim.opt.relativenumber = false
```

Find mode

If you need to move your cursor to a position that is relatively close to its current position, you may want to use LazyVim's Find mode instead of the Seek mode we described earlier. The default Find mode in Neovim is rather limited, but the `flash.nvim` plugin that enables Seek mode makes it much nicer to use.

To enter find mode, press the `f` key. Like Seek mode, a portion of your screen will dim, indicating that you should type another character, and after you do so, all instances of that character *after the cursor* will be highlighted.

This is where the similarities between Find mode and Seek mode end, however. My cursor (which was originally in the middle of the word 'described' in this screenshot) immediately jumped forward to the first `s` (case insensitive) in the document. You'll also notice that none of the text before the place where my cursor has been dimmed, and that none of the `s` characters in the lines before my cursor have been highlighted.

Also unlike Seek mode, there are no labels to jump directly to any of the `s` keys that have been highlighted, and I cannot type additional search characters to narrow down the search.

Instead, I need to use counts to jump to later instances of `s`. If I want to jump ahead to the third highlighted `s`, I type `3f` and my cursor will move there. However, if you want to jump to a much later `s`, you probably don't want to individually count how many `s` keys there are. Luckily, after you use a count, LazyVim leaves you in find mode, so you can just guess how many `s` characters there are and then once you are closer, enter a new count. If you only want to jump ahead by one `s` character, you don't need to enter a count, just press `f` by itself and you'll move ahead.

If you miscounted or misguessed and jump too far, fear not! You can take advantage of the fact that `Shift-F` means "find backwards", and can also be counted. So if you need to move to the 15th highlighted `s`, it's totally fine to guess `18f`, realize you've gone three too far, and use `3<Shift>F` to jump back to the previous character.

Moreover, if you know that the character you are looking for is behind or above your cursor in the document, you can enter find mode with `Shift-F` instead of `f` in the first place. This will immediately start a backwards find operation instead of a forward one. And if you know right off the bat that you want to jump back or ahead by three instances of the given character, you can even use a count when you first enter find mode.

There is also a subtle variation of Find mode that I call “To” mode, although the official Vim mnemonic is actual “til” mode. You enter it with a `t` or `Shift-T` depending on what direction you want to go.

“To” mode behaves identically to find mode except that it jumps to *just before* the target character.

You might think that to mode is kind of redundant because you could fairly easily use find mode followed by a single `h` to move the cursor left. But “To” mode is extremely useful when you are combining it with operations to edit the text, which we will discuss later. As a taste, if you use the command `d2ts`, it will delete all text between the cursor and the second `s` it encounters, but leave that `s` alone. This is than the `d2fsis<Escape>` that would be required if you used a find command and then had to enter insert mode to add the `s` back.

Moving by Words

When `f` or `t` feels too big, and cursors with counts feel too small, you’ll most likely want to use the word movement commands. In other editors and IDEs you might be used to getting this functionality by holding `Control` or `Alt` (depending on the operating system and editor) while using the arrow keys.

Neovim is easier; you don’t have to move your hands to the arrow key section of the keyboard and you don’t have to hold down multiple keys at once.

Instead, you can just enter normal mode and press the `w` key to move to the beginning of the next word. If you instead want to move to the *end* of the current word, use the `e` key. If you are *already* at the end of the current word, `e` will go to the end of the *next* word. This is useful when you want to combine it with counts: If you need to move to the end of the word that is two words after the current word, press `3e`. This is the same as pressing `e` three times, which would move to the end of the current word, then the next word, and finally to the end of the word you want to hit. `w` can also be prefixed with a count if you need to move to the beginning of a word that is a certain number of words after the current one.

Use the `b` key if you want to move backwards instead. This will move you to the beginning of the current word, or if you are already at the beginning of the word, it will move to the beginning of the previous word. As usual, use a count to move even to the beginning of even more words.

Surprisingly, it takes a bit more work to move to the end of the *previous* word, as you need to press two keys: `g` followed by `e`. The mnemonic for this is “**g**o to **e**nd of previous word”. In practice, you’ll find that you hardly ever need this functionality for some reason, and the honest truth is I usually use `be` (`b` to move to beginning of previous word, then `e` to go to end of that word) to move to the end of the previous word. If you do use `ge`, however, it can be combined with a count as well. You’ll need to type something like `4ge`, depending on the count. The command `g4e` wouldn’t do anything useful.

Collectively, you may occasionally hear the `w`, `e`, and `b` commands referred as the “web” words. It just means “moving by words”. These are probably the most common movements you will use, more than individual cursor positions, simply because most editing actions tend to involve changing or deleting a word or sequence of words.

Moving by Words, Only BIGGER

The “shifted” form of the web words also move by words, but the definition of “word” is different. Specifically, a capital `W` will move to just after the next whitespace character, where a lowercase `w` will use other forms of punctuation to delineate a word. Consider a method call on an object that looks something like this in many languages:

```
myObj.methodName( 'foo', 'bar', 'baz' );
```

If your cursor is currently at the beginning of that line, a `w` will move your cursor to the period on the line, a second `w` will move you to the `m`, and subsequent `w` presses will stop at the paren and quotes as well.

On the other hand, if your cursor is at the beginning of the line, a `Shift-W` will move you all the way to the first quote in the “bar” argument, since that is where the first whitespace character is.

As a visualization, here are all the stops on that line of code when you use `w` compared to when you use `W`:

```
myObj.methodName( "foo", "bar", "baz" )
-----ww-----w-w--w--ww--w--ww--w----->
-----w-----W-----W----->
```

The `B`, `E`, and `gE` motions behave similarly, moving in the appropriate direction by whitespace-delineated words instead of punctuation ones.

One thing that is kind of annoying both in Vim and the way LazyVim is configured is that there’s no way to navigate between the individual words of `CamelCaseWords` or `snake_case_words`. You can use `fC` or `t_` and similar if you want to, but I will later show

you up how to set up the `nvim-spider` plugin that makes navigating these common programming constructs simpler.

Line targets

Very frequently, you need to move to the beginning or end of the line you are currently editing. Often you can use `I` or `A` for this if your goal is to move to that location and enter insert mode, but if you need to move there and stay in normal mode (e.g. for other purposes such as to delete or change a word) you can use the `^`, `$`, and `0` commands.

If you are familiar with regular expressions, you might know that `^` is used to match the start of text or start of the line and that `$` is used to match the end, so the mnemonic of using these two keybindings to match the beginning and end of the current line will hopefully be less unmemorable than they seem at first.

There is a certain lack of symmetry between the two, however. The `$` (`Shift-4`) command simply means “go to the end of the line”, as in the last character before the ending newline, no matter what that character is. The `^` or caret (`Shift-6`) means “go to the beginning of the text on this line”. The “of the text” there is important: if your line has whitespace at the beginning (e.g. indentation), the `^` caret will **not** go to the very first column, but will instead go to the first non-whitespace character.

To move to the very beginning of the line, use the `0` key. `0` is the **only** numeric key that maps to a command because the others all start a count. But it wouldn't make sense to start a count with `0`, so we get to use it for “move to the zeroth column”.

There is also a command to go to the end of the line excluding whitespace, but I have never used it, probably because I usually have formatters configured to trim trailing whitespace so it doesn't come up.

The two character combination `g_` (`g` underscore) means “go to the last non-blank character”. I guess `_` kind of looks like “not a space”, so it's kind of mnemonic? I include it to be comprehensive, but you'll likely not use it much. You also have the option of combining other commands you've learned so you don't have to memorize this one off. For example, you can use the three character `$ge` (combining “end of line” with go backwards to end of word) or `$be` to move to the last non-blank on the line. You have options; pick the one that you find is easiest to remember or type!

Jumping to specific lines

If you compile some code or run a linter, you will invariably be given a line number where the error occurred (unless the compiler is particularly useless).

You can jump to a specific line by entering the line number as a count, followed by `Shift-G`. So `100G` will move your cursor to line 100.

`Shift-G` is a normal command, though, so you can issue it without a count, in which case the `G` command will always take you to the end of the file.

You can go to the top of the file with `1G` if you want, but since this is such a common operation, you can instead use `gg` (two lower case `gs`). The mnemonic for `g` in all cases is “Go to”, and there are a lot of things that can come after a `g` (`:help g` will introduce you to the ones I don’t cover, although be aware that LazyVim has overridden some of them).

Since the most common place you are likely to want to “go to” is a line number, the easiest to type `G` and `gg` commands are used for line number navigation.

Jump History

All this jumping around can make you feel a little lost. Luckily, there are two super-useful keybindings for going back to places you previously jumped.

`Control-o` is the non-modal control-based keybinding that I use most often. I should honestly bind it to something more accessible, I use it so much. It basically means “Go to the place I jumped from”.

This is super handy when you’re editing code deep in a file or module and realize you need to import a library at the top of the file. You can use `gg` to jump to the top of the file, `s` to seek to the line you want to add the import on, and then enter insert mode to add the import. Now you want to go back to the code you were working on so you can actually use the import. `Control-o` a couple times will take you there.

NeoVim keeps a history of *all* your jumps, so you can jump between several locations (perhaps to look up documentation or the call signature for a function) and always find a way back.

If you jump too far, you can use the `Control-i` keybinding to jump *forward* in history. It’s just the opposite of `Control-o`. I don’t know why `i` and `o` were chosen for these; maybe because they are side-by-side on a Qwerty keyboard? They are used commonly enough that once you learn them, you won’t forget.

Summary

Navigating code is a huge topic in Vim. You’ve already learned enough commands that you can navigate a Vim window more efficiently than most non-modal editors can dream of. But

we've actually barely scratched the surface, and we'll be covering a bunch of even more useful code navigation commands in a later chapter.

We covered the LazyVim `seek` mode to jump anywhere in the visible window, and then the scrolling commands to make sure the thing you want to jump to is visible. Then we covered moving the cursor with the home row key and extended them with counts.

We learned how Find mode differs from Seek mode, even though they are superficially similar. Then we covered some standard keybindings for moving by words and to key places on a line before jumping to specific lines. We wrapped up by covering how to navigate to places you have jumped before.

In the next chapter, we'll learn more about opening files and navigating the Filesystem.

Chapter 4: Opening Files

In the previous chapter, as a side-effect of learning about command mode, we saw how to open files the old-fashioned Vim way, using the `:edit` command. Another old-school alternative is to open them directly from the terminal shell command line, using `nvim filename`.

Both of these are occasionally handy, but LazyVim pre-configures a few more modern ways of navigating and opening files.

Introducing Telescope

Telescope is a wonderful plugin whose functionality should, in my opinion, be built into Neovim by default. Since it isn't, we have to rely on the fact that it is shipped with LazyVim by default instead!

Telescope is essentially a "picker" interface with preview and fuzzy search capabilities. If you've used the command menu in many modern editors (or even Github or Slack), you may know what I'm talking about. Telescope itself doesn't care what you are picking, and there are a wide variety of plugins available that use it for many different tasks. Telescope ships with the most common aspects.

The most common task you will perform with Telescope is to open a file using fuzzy search. I use this command dozens, maybe hundreds of times per day, so it's a good thing it's got a really accessible keybinding.

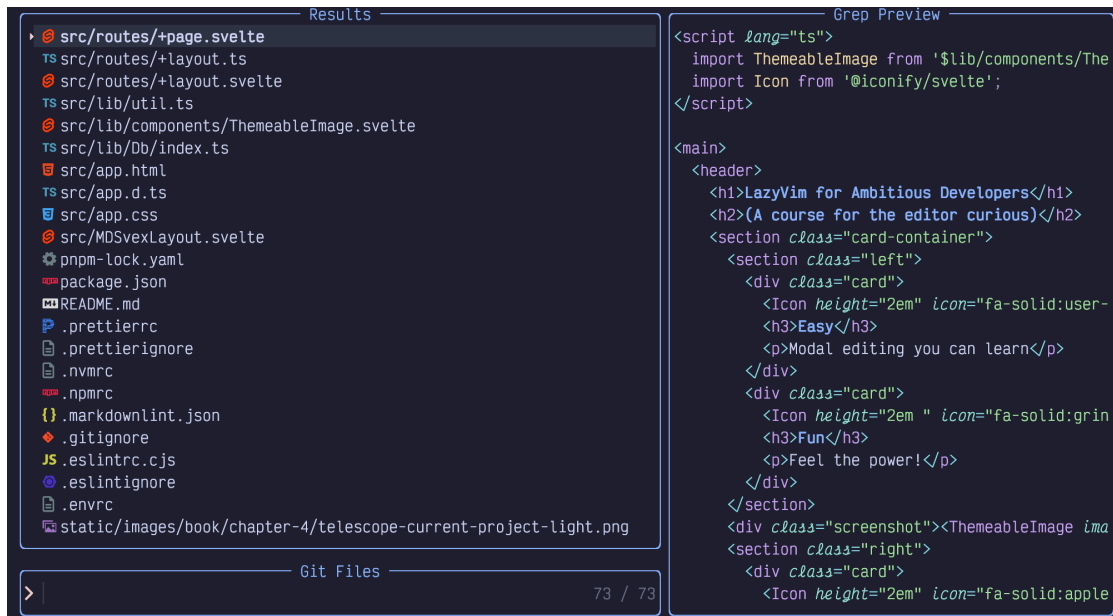
The Telescope file picker is best illustrated while working in a code repository with a lot of files. So close Neovim with `Space q q` and use the `cd` command on your terminal to

change to the directory of a project you've been working on recently (If you don't have one close to hand, clone your favourite open source project and use that instead). Then type `nvim` to open Neovim again.

Tip: I had you exit to the terminal above because it's easy to reason about, but it is also possible to change directories from inside LazyVim using the `:cd` command. Type `:cd the/path/to/the/directory` and hit enter, remembering that you can use the Tab key to autocomplete the path. Now if you use `:e` to open files, they will be relative to the directory you specified. If you are using Telescope, they may be relative to that cwd or to the project containing the current file, as discussed shortly. Use `:pwd` to see what the current directory is.

Ok, so you're in the root directory of a large project and you want to open an arbitrary file. Simply press Space twice (i.e. Space Space) to pop up the "Files In Current Project" telescope picker. As I mentioned, this is the easiest keybinding to type on your entire keyboard. The Space bar on most keyboards is big, and you're hitting it with your strongest digit – the thumb. As usual, just one Space will pop up the Space mode menu, and you can see that a second Space will present you with "Find Files (root dir)".

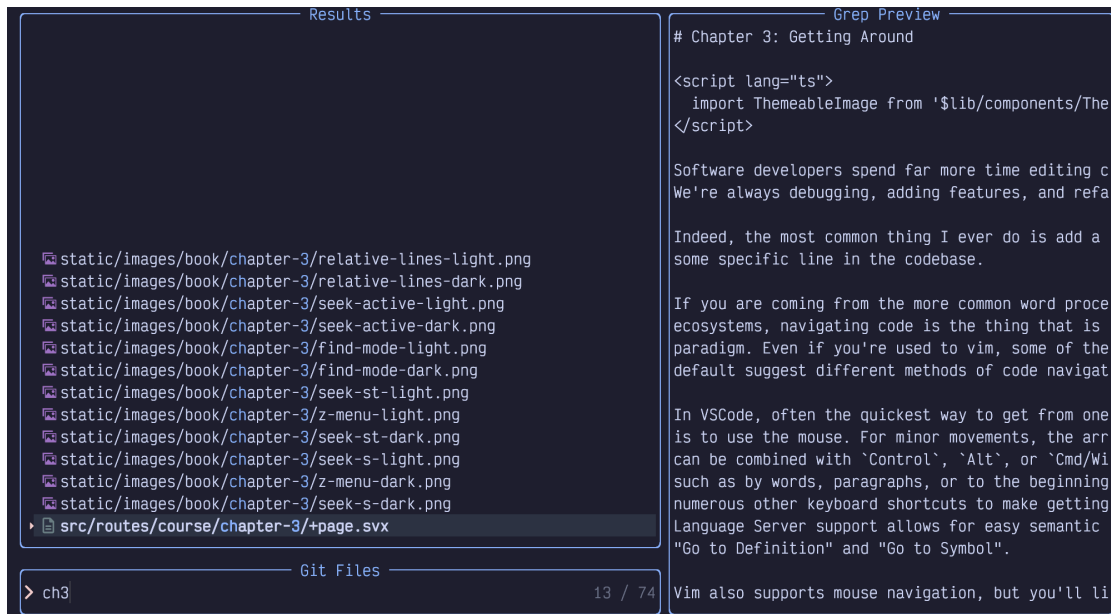
For the project containing the current state of this book, the picker looks like this:



screenshot

The picker is divided into three main areas: the results list in the upper right, a preview of the currently selected file on the right, and the Input area, in this case labelled "Git Files".

The input area is actively focused and currently in insert mode, so you can just start typing the name of whatever file you want to open. This is a “fuzzy search”, (a concept popularized by Sublime Text) which means you can skip letters, saving you oh-so-precious milliseconds. For example, if I type `ch3`, my list gets filtered down to the following files:



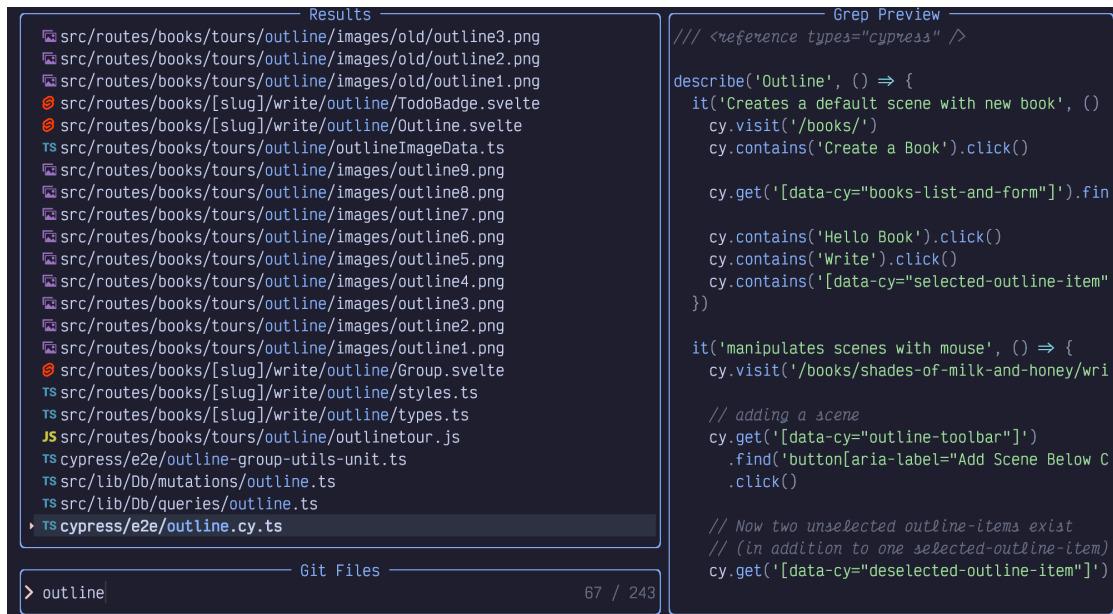
screenshot

Only files whose paths contain those three characters in order, with possibly other characters in between, are visible. Telescope has helpfully highlighted those three letters in the results so you can easily see why it matched (Though, depending on the medium you are reading, it may not be clear in the image.)

Also notice that by default, the match is case **ins**ensitive. I typed the lowercase letter `c`, but it matched the uppercase `C` in the filename. This is usually sufficient to narrow the search results to what you need. However, if you *do* use **any** capitalized letters in your search than it switches to a case sensitive mode (this is sometimes referred to as “smart case”).

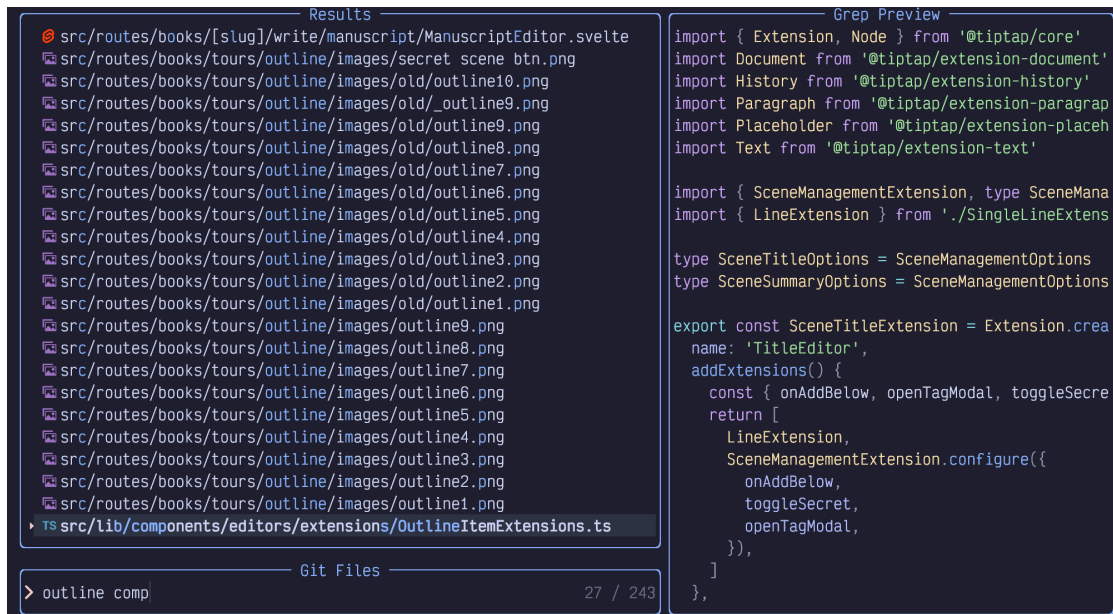
That means that `Ch` will match all the `Chapters`, but `cH` will not match anything at all. More interesting, `chF` will *also* not match anything at all because the presence of the capitalized `F` makes the whole thing case sensitive, and the chapters are all named with a capital `c`, so the lowercase `c` is not able to match them.

Another neat Telescope matching trick: Sometimes you will start typing a word and realize you need to match something *earlier* in the path to distinguish it. For example, I started typing `outline` in these source files from [Fablehenge](#):



screenshot

Outline is a common word in this app. There are 243 matching files, and I realize I should probably have typed `comp` in front to narrow it to just files in the `component` directory. I *could* switch to normal mode and edit the beginning of the line, but it's faster to just type `<space>comp`. Telescope will interpret the space as "filter the lines again fuzzy matching this new word from the beginning". Here we can see that only `comp...outline` files have been matched:



screenshot

This image might be a bit surprising; the most promising match is obviously the one at the bottom of the list (which is why it is selected). The other 27 matching lines contain all the letters of the word “outline” and all the letters of the word “comp” in order from left to right. However, because of the fuzzy matching algorithm, the two can actually overlap! So on e.g. the second-to-last entry, the `c` of the matching “comp” is *before* the word `outline`, the `o` is *in* it, and the `m` and `p` both come *after* the word `outline`. Telescope doesn’t care, though it will rank matches with the matching letters closer together as more important, so they’ll be visible at the bottom of the results.

You can use the up and down arrow keys to select a different file in the search results, and its preview will show up in the right-hand window. Once you find the file you want to open, press the `Enter` key to open it in the currently active Neovim window.

The Telescope input area even has its own normal mode! You can get into it using a *single* press of the `Escape` key. Now if you press `j` or `k`, you’ll be able to select different files in the list without moving your hand to the arrow keys. Further, the `h` and `l` keys will allow you to move the cursor within the input box and you can use the `i` or `a` keys to enter insert mode at the new location. The “but bigger” `I` and `A` keys allow you to move the cursor to the beginning or end of the line and enter insert mode as well.

You can even use seek mode, as we discussed in Chapter 3, though it works a bit differently. When you press the `s` key while in the telescope picker’s normal mode, you can skip the

part where you enter a character to search for. Instead, LazyVim will immediately label every line in the picker with a character to the left of the filename:



screenshot

These characters are labels for each line in the picker. Simply press one of the shown letters on your keyboard, and whichever line the label associated with that letter is on will be selected. Then press `Enter` to actually open the file.

Finally, if you are in the Telescope window and decide you don't want to open any files after all (or you got the information you needed from looking at the preview and therefore don't need to open it all the way), press `Escape` *twice*. Once to enter normal mode in the Telescope picker, and a second time to close the picker.

If you need to scroll the *preview* window to see something lower down in the file, the same `Control-d`, `Control-u`, `Control-f`, and `Control-b` keys that we discussed in the Basic Navigation chapter can be used.

The difference between “Root” and “cwd”

The `<Space><Space>` command is mapped to “Find Files (Root Directory)”. Two other ways to open telescope are to use `<Space> f` to open the “file/find” menu, and either `f` or `F` again.

`<Space>ff` is the same as `<Space><Space>`. It opens “Find Files (Root Directory)” and is just another longer way to get there. I assume it exists in both places so that users can choose to map some other action to `<Space><Space>` and still be able to access that functionality; `<Space><Space>` is the easiest keybinding to hit on the keyboard, so it makes sense to assign it to your most common action. If that action isn’t opening files with Telescope, you will still want to be able to do that with the slightly longer `<Space>ff` keybinding.

`<Space>fF`, where the second `F` is shifted, is similar; it is mapped to an action called “Find Files (cwd)”. If you run it in your project, you’ll probably find that it appears to do the exact same thing as “Find Files (Root Directory)” (depending on how your project is set up), so the purpose of two separate keybindings may be confusing.

Current Working Directory

`cwd` stands for “Current Working Directory”, and by default, it refers to whatever directory your terminal was in when you typed `nvim` to open the editor. As I mentioned briefly while discussing tab completion in the command menu, you can change the `cwd` for the entire editor by entering command mode with `:` and then typing `cd path/to/directory` (remember, all commands are followed by a carriage return, so press `Enter` or `Return` afterwards). Now if you use `<Space>fF`, the list of files will be shown relative to the new directory you have changed into.

If you are unsure what directory you are in, you can use the `:pwd` (short for “print working directory”) command to have it pop up in a little notification window. `cd` and `pwd` are the same commands used by `bash`, `zsh`, and many other shells for changing and printing the working directory, so they may already be familiar to you.

We haven’t discussed splitting your editor or opening new tabs yet, but this is a good time to note that it is actually possible to have *different* working directories for different windows. The command to change just the current window’s directory is `:lcd`, short for “local change directory”. This can be a powerful way to work on multiple projects at the same time (for example, if you are a full stack developer working on backend and frontend projects). However, the LazyVim concept of a “Root” directory can semi-automate a lot of this.

Root directory

The root directory is not a Vim concept, but is instead a Language Server Protocol (LSP) concept. LSPs are the reason that VSCode became so popular so quickly; the idea was that the editor could call out to an external service running on your computer to find out useful things about the codebase. The LSP powers a lot of useful stuff such as go to definition and

references, highlighting errors in your code, and showing documentation for a variable or class. It can even help with formatting and syntax highlighting!

The root directory is the directory that the LSP infers is the “home” directory of the currently open file. How the LSP does this is language (and language server) dependent. For example, in Javascript or Typescript projects it probably searches parent directories for the presence of a `package.json` or `tsconfig.json` file to detect the root directory, whereas in a Python project it might instead look for things like `pyproject.toml` or `poetry.lock`, and Rust projects use the directory that contains a `Cargo.toml`. Or the LSP might just use the presence of a `.git` folder as the “root” of the project’s workspace.

The only reason this root directory is “often the same as your `cwd`” is that this is usually the folder you want to work from when you are working on a project, so it’s the one you `cd` into before you open Neovim.

This automatic root directory thing can be super useful if you are working on multiple projects. Instead of using `lcd` as discussed in the previous section, you can just open a file in a different project using `:e` or one of the file finding extensions we’ll discuss next. Then if you invoke the “Find files (root dir)” command using `<Space><Space>` or `<Space>ff`, it will look for other files in the same root directory as the one you just opened.

However, it can sometimes be confusing, especially if you are working in a so-called monorepo or if you have root directories in places you don’t expect. For example, I have a fairly normal Svelte project that has a `package.json` file in it. This project uses Cypress for testing, and the Cypress folder has a `tsconfig.json` file in it that causes the Typescript language server to interpret that as a separate root. So if I am working on one of the cypress test files and press `<Space><Space>`, the root directory is considered the Cypress folder and I can only open other cypress tests. But often the thing I *wanted* to do was open a source file in the main folder to see why a test is failing. In this case, I have to press `<Escape><Escape>` to exit the Telescope picker, then `<Space>fF` to open the picker in current working directory mode instead.

Telescope file finder isn’t the only LazyVim tool to use the related concepts of working and root directories, and we’ll discuss two others next.

The Neo-tree.nvim plugin

Neo-tree creates a left-sidebar file explorer experience that should be familiar to users of many modern IDEs and editors. While, like many of those environments, Neo-tree does work with the mouse, it is optimized for keyboard interactions, making it faster to work with once you learn “Neo-tree mode”.

I want to be upfront and honest here: I don't personally use Neo-tree. I find that the Telescope picker is the fastest way to open files, and when I need to manipulate the filesystem, I prefer to use `mini.files`, which we will discuss shortly. The primary reason I prefer `mini.files` is that it uses the same keybindings as vim normal mode. Modes are great, but having more than necessary is not!

However, over my lifetime, I have received plenty of hints that I may be rather weird! I suspect that many readers will prefer the familiar tree view experience Neo-tree provides, and since this plugin ships with LazyVim by default, I want to make sure it gets fair coverage in this book.

Let's start by opening Neo-tree using the `<Space>-e` keybinding, where the mnemonic is "e for Explore". If you pop up the space mode menu, you'll see that, as with telescope, there are two ways to open the Neo-tree explorer: `<Space>-e` for Explore Neo-tree (root directory) and `<Space>-E` for Explore Neo-tree (cwd).

"Root directory" and "cwd" have the same meaning we discussed in the previous section, and you will notice the consistent relationship between lowercase and uppercase letters: `<Space>ff` and `<Space>e` both open the root directory, and `<Space>fF` and `<Space>E` both open the current working directory.

Tip: To hide the explorer window, just press `<Space>e` again while it is visible, or press `q` while the explorer window is focused.

When the explorer is opened, it shows all the files and folders in the relevant directory, with all the folders collapsed, except for the one containing the currently active file, if there is one. For example, while editing this file, my Neo-tree looks as follows:



screenshot

It may not show up clearly in the screenshot, but the cursor is on the file I'm currently editing. I can move that cursor up and down using the arrow keys or the ubiquitous `j` and `k` keys.

Folders are collected to the top of the view. If you move the cursor to one of these folders, you can press the `Enter` key to expand the folder. And if you move it to a file, you can open the file in the current vim window with the same `Enter` keypress.

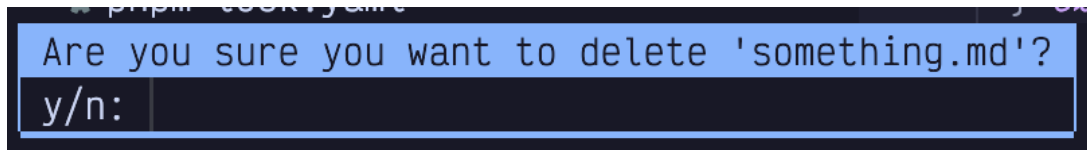
You can also expand and collapse folders and open files by double clicking with the mouse, but my guess is you won't want to do that once you learn proper keyboard navigation.

Speaking of keyboard navigation, yes, `j` and `k` to move up and down can be super slow if there are a lot of files to navigate. All of the commands that we discussed in Chapter 3 can be used to move faster. For example, `10j` will move the cursor 10 lines down with just three keystrokes compared to pressing `j` 10 times, and `Control-d` or `Control-u` can be used to scroll the tree down or up. Most interestingly, `s` can be used to Seek to any line in the Neo-tree view, even if Neo-tree is not currently focused.

Neo-tree will show the root or `cwd` as the topmost directory. If you need to navigate "up" the tree to a higher-level directory, you will need to use the `Backspace` key.

Tip: Backspace is often coded as <BS> in Vim, so if you see a keybinding or instructions telling you that <BS> does something, they aren't full of (bull)! It just means Backspace.

In addition to navigating and opening files, you can even make changes to the file system using Neo-tree. For example, to delete a file, you can move the cursor over that file and hit the `d` key. You'll be prompted with a popup window asking if you are sure; hit `y` and then Enter to confirm it:



screenshot

To add a file or folder/directory, use the `a` key and enter a new name. Use a trailing slash (`/`) to indicate a folder. You can also use the `A` key in the explorer to add a folder without having to type a trailing slash.

The `r` key can be used to rename the or folder under the cursor.

To copy or move a file, you can use Neo-tree's pseudo-clipboard. I say "pseudo-" because you can't use this to copy a file to be pasted in MacOS Finder or Windows Explorer; only to other places in the Neo-tree.

To *cut* a file with the intent of moving it somewhere else in the tree, use the `x` command. If, instead, you want to *copy* the file, use `y`. The mnemonic for `y` is yank, and is actually the same key you would use to copy text in the normal editor. To complete the move or copy, you'll need to navigate to the folder you want to move or copy it into and use the `p` key (which you may recall means "put" or "paste").

Neo-tree also has a Filter mode that I find quite clumsy; it's really just a cheap imitation Telescope picker in a smaller window, so I recommend using the Telescope picker instead. You can access it using `/` and enter some characters to limit the search results to files that match those characters. Then use the up and down arrows to navigate the list (`j` and `k` won't work here because you're in a sort of Insert mode context).

There is a *ton* of other cool stuff that Neo-tree can do. We will cover its use for buffer, git, and symbol navigation later, for example. In the meantime, you can use the `?` (mnemonic "ask question for help") key while the Neo-tree window is focused to get an overview, `:help neo-tree` if you want to drink from the firehose.

The `mini.files` alternative

As I mentioned, I don't actually use Neo-tree for file navigation. I find that it feels kind of "foreign and un-vim-like". To me, it is a completely separate experience that just happens to be embedded in a vim window. That said, I *also* don't like the tree view sidebar experience in VSCode and the editors it emulates / is emulated by, so it's possible that tree views just aren't right for me.

These are just **my** opinions, and one of the golden rules of text editors is "all opinions are valid" (otherwise there would be war). A large number of Neovim users love Neo-tree, and you should use it if it matches your mental model.

That said, I'm clearly not alone in these opinions, because LazyVim optionally provides a different file management experience called `mini.files`. It is disabled by default.

`mini.files` is part of a suite of fairly random Neovim packages known as `mini.nvim`. These plugins are largely independent from each other and provide a lot of common features that in many cases ought to ship with NeoVim. In some cases, the `mini.nvim` plugins are inferior to other plugins that they clone, but a number of them are best in class. `mini.files` is not the only `mini` plugin that ships with LazyVim, and we'll touch on others later.

The `mini.files` file manager is kind of like a Neovim-native experience of the columnar view that is popular in MacOS's finder, among other file managers. The main reason I like it is that editing the directory listing is just like editing a normal text buffer. I don't have to remember that `a` means "after" in Normal mode, but it means "add file/folder" in Explorer mode. Instead, in `mini.files`, I use the `o` key to "create a new line below the current line", and then enter the file name in Neovim Insert mode. Later, I tell `mini.files` to sync my changes and it will create the file for the new row.

In order to use `mini.files`, you have to enable it as a *Lazy Extra*. I go into more detail about Lazy Extras in the next chapter (When I told her the first draft of this chapter was over 8100 words long, my editor said "Dear Lord lol"), but the basic instructions are:

- Type `:LazyExtras<Enter>`
- Move your cursor to the line that contains `mini.files` (Seek mode is fastest)
- Press `x` to install the **eXtra**
- Wait a moment for the plugins to install
- Restart Neovim

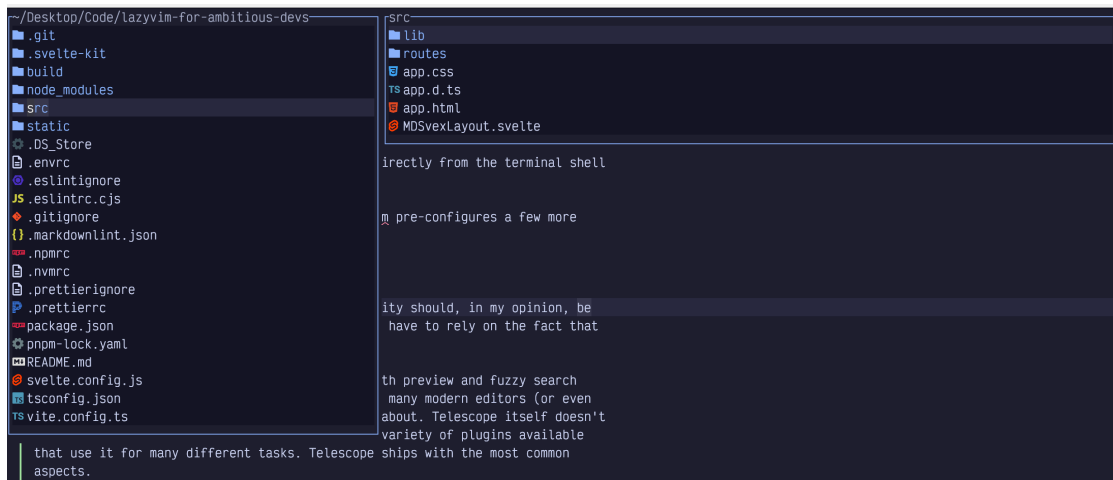
Using mini.files

Once installed, you can show the mini.files view using `<Space>fm` and `<Space>fM`. By default, these are *not* quite the same as the `cwd/root` structure we've seen in Telescope and Neo-tree. Instead, they are listed in the `<Space>f` menu as follows:

```
m -> Open mini.files (Directory of Current File)
M -> Open mini.files (cwd)
```

The default `mini.files` configuration doesn't have an open in root option. I like having the ability to open the directory of the currently open file, but I don't like *losing* the ability to open the root of the current project. I show how to address this in the next section where we discuss keybindings.

Instead of a sidebar, the `mini.files` menu shows up as columns of windows (known as Miller columns) side-by-side. For example, here's what happens when I open `mini.files` to the current working directory of this book:



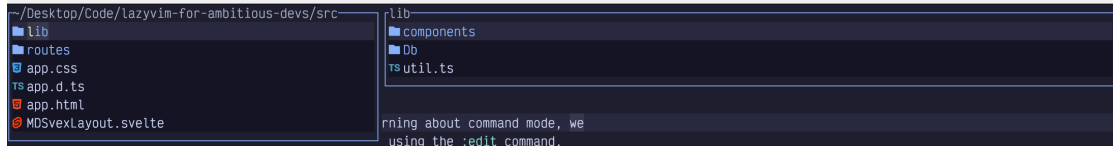
screenshot

This book is published as a svelte-kit app. The left-hand pane shows the current working directory, and the right pane shows the contents of the `src` directory, which is the “focused” folder in the left pane.

Interacting with `mini.files` is *very* similar to interacting with a standard vim window. You can use the `j` and `k` keys to move the cursor up and down. If this places your cursor over a folder, the contents of that folder will immediately show up to the right, and if it is over a file, you will see a preview of the file (by default, the previews are smaller than what

I have in these screenshots; I'll show you how to change that in the next section if you have the screen real estate for it).

If you want to move “into” a folder to interact with the contents of that folder instead, simply press the `l` key to move “right”. Here, I moved my cursor into the `src` folder, which immediately opened the file under the cursor in a new preview window.



screenshot

Similarly, pressing `h` will move “out” of the current folder. If the cursor is in the left-most column, moving left will open a new left-most column, so you can navigate right up to the root of your file-system if you need to.

To open a file in the currently active Neovim window, press `l` on that file again. The behaviour here may be a bit surprising; the file will open *under* the `mini.files` view, but it won't hide the file menu. This allows you to open multiple files before closing the navigator (which can be done with the `q` key).

The beautiful thing about `mini.files` compared to `Neotree` is that the little windows act like normal editors, and all the navigation features you have become used to are available. For example `seek` mode can be used to navigate to a file. Press the `s` key and then any number of characters you want to search for. Any matches to the typed characters will be labelled and you can jump to them by typing the indicated label.

Even modifying the filesystem is exactly the same as editing a normal buffer. We haven't really covered editing yet (I'm just as surprised as you are), but here's a quick overview:

- To rename a file or folder, navigate to the line that has it, and enter insert mode to change or add text.
- Deleting a file or folder uses the command `dd` which is the keybinding to delete an entire line of text in normal Neovim windows.
- Copy a file or folder with `yy` the command to copy (yank) a line of text
- Put/paste a deleted **or** yanked file with `p`.

We'll discuss these commands and more in Chapter 6. The main point is that pretty much any navigation or editing command you learn in the future will work with `mini.files`.

Saving Filesystem Changes

Any modification that you make using these keybindings will not actually be saved on the filesystem until you type the = key, which is a (rare) `mini.files` specific keybinding. I think of it as meaning “make the filesystem *equal* to what I’ve typed”. This will pop up a little window telling you what actions `mini.files` wants to take on your behalf, such as deleting, moving, renaming, or copying files. You can confirm or decline the changes with a `y` or `n` (yes or no, of course).

I encourage you to play with both Neo-tree and `mini.files` until you can make a decision as to which of the two you prefer. Eventually, you will arrive at one of the following conclusions:

- You prefer Neo-tree and don’t need `mini.files`. In this case, revisit the LazyExtras mode and disable `mini.files` with the `x` key.
- You use Neo-tree for some interactions (possibly things we haven’t covered yet, such as navigating git, buffers, or symbols) and `mini.files` for others. In this case, you are probably content with the default LazyVim configuration of the `mini.files` extra.
- You are *my kind of weird* and don’t want to use Neo-tree at all, preferring only `mini.files`. This exact situation is discussed in the next chapter as we learn more about configuring plugins.

Summary

In this chapter, we learned not one, but three different ways to open files and interact with the filesystem in LazyVim: Telescope, Neo-tree, and `mini.files`. Each provides a different mechanism for opening files, and you will find some of them more comfortable than others.

As a side-effect of studying these filesystem tools, we learned a little bit about configuring plugins and installing LazyVim extras. we will go into more detail of this in the next chapter.

Chapter 5: Configuration and Plugin Basics

I’ve mentioned plugins a few times previously and you even got to see the `lazy.nvim` plugin manager in action back in Chapter 1. LazyVim has a unique multi-layered approach to managing plugins that requires a bit of description, but is very elegant in practice.

Installing plugins allows you to configure NeoVim to do things it can’t do by default. Plugins are written in either Lua or VimScript (though most NeoVim users prefer Lua-based plugins).

The Three Categories of Plugins in LazyVim

The simplest plugins to use in LazyVim are pre-installed by LazyVim itself. You've used many of them already. Some, such as Neo-Tree, Telescope, and lazy.nvim provide custom UI components to interact with them. Others, such as flash.nvim and which-key provide new commands or modes to work with. Still others operate quietly in the background auto-matching parenthesis or tags and drawing indent guides.

These plugins are preconfigured in LazyVim with sane defaults. Because they are so well integrated, customizing those defaults is doable, but sometimes requires a few tricks that we will cover in this and later chapters.

The second category of plugin in LazyVim are the so-called "Lazy Extras". These plugins are **not** enabled by default, but can be enabled with just a couple of keystrokes if you want them. Lazy Extras exist to make it easy to install popular plugins with a configuration that is guaranteed to play nicely with the other plugins that ship with LazyVim.

The third category includes third-party plugins that LazyVim has no awareness of. You will have to configure these plugins from scratch and do your own due diligence to ensure that keybindings and visual artifacts don't conflict with the plugins that LazyVim manages. In a non-LazyVim configuration, all plugins fell in this category, and it could be a headache to maintain as plugins evolved and fell out of use over time. In LazyVim, relatively few plugins fall in this category, so the whole experience is much more pleasant.

As a specific example consider these three Neovim plugins for file management, two of which we discussed in the previous chapter:

- `Neo-tree.nvim` ships with LazyVim and is active by default. The LazyVim configuration for Neo-tree does not conflict with other LazyVim plugins by default.
- `mini.files` ships as a Lazy Extra, and is basically a "one click" (or, since this is vim we're talking about, one keypress!) install that is guaranteed to work with LazyVim.
- `oil.nvim` is an alternative plugin for filesystem management that LazyVim does not explicitly support. You can install it in LazyVim with a few lines of configuration, but it's not quite as easy to set up as `mini.files` and there is no guarantee it won't have conflicts you need to sort out yourself.

From NeoVim's point of view, all these plugins are exactly the same, as NeoVim only knows about third-party plugins. LazyVim just comes with a bit of extra structure that you need to think about when using plugins. Usually this structure simplifies things, but sometimes it gets in the way.

Lazy Extras

In the previous chapter, I covered how to use `mini.files`, but I was pretty terse on the installation instructions. Now we'll get to dive deep.

The Lazy Extras mode can be accessed by pressing `x` from the dashboard. If you aren't on the dashboard, you'll need to enter command mode with `:` and type `LazyExtras` followed by the usual `Enter` to confirm a command (Incidentally, you can also show the dashboard at any time by typing the command `:Dashboard`).

Either way, you'll be presented with a list of possible plugins to install. On my setup this looks as follows:



```
LazyVim Extras

This is a list of all enabled/disabled LazyVim extras.
Each extra shows the required and optional plugins it may install.
Enable/disable extras with the <x> key

Enabled: (13)
• coding.copilot ④ copilot-cmp ④ copilot.lua ④ nvim-cmp ④ lua-line.nvim
• coding.yanky ④ sqlite.lua ④ yanky.nvim
• editor.mini-files ④ mini.files
• editor.trouble-v3 ④ trouble.nvim ④ edgy.nvim ④ lua-line.nvim ④ telescope.nvim
• formatting.prettier ④ mason.nvim ④ conform.nvim ④ none-ls.nvim
• lang.go ④ mason.nvim ④ neotest-go ④ nvim-dap-go ④ nvim-lspconfig ④ nvim-treesitter ④ conform.nvim ④ neotest ④ none-ls.nvim ④ nvim-dap
• lang.markdown ④ headlines.nvim ④ markdown-preview.nvim ④ mason.nvim ④ nvim-lspconfig ④ nvim-treesitter ④ none-ls.nvim
④ nvim-lint
• lang.python ④ neotest-python ④ nvim-cmp ④ nvim-dap-python ④ nvim-lspconfig ④ nvim-treesitter ④ venv-selector.nvim ④ neotest ④ nvim-dap
• lang.tailwind ④ nvim-cmp ④ nvim-lspconfig ④ tailwindcss-colorizer-cmp.nvim
• lang.typescript ④ mason.nvim ④ nvim-lspconfig ④ nvim-treesitter ④ nvim-dap
• linting.eslint ④ nvim-lspconfig
• util.mini-hipatterns ④ mini.hipatterns
• util.project ④ project.nvim ④ telescope.nvim ④ alpha-nvim ④ dashboard-nvim ④ mini.starter

Disabled: (41)
```

screenshot

I've installed over a dozen extras at the moment, mostly for the various programming languages I dabble in. You can navigate this file using all the standard navigation commands such as `j` and `k s`.

No matter how you get there, once your cursor is on the extra you want to install (such as `editor.mini-files`) line, just hit the `x` key to install the extra. If you want to uninstall it, do the same thing; move to the appropriate line (now under the list of `Enabled` extras), and hit `x` to disable the extra. The mnemonic here, of course is that `x` means "Extra".

You may need to quit and restart Neovim for `lazy.nvim` to pick up that the extra has been installed and sync its dependencies.

While we're in the `LazyExtras` screen, I recommend enabling the `lang.*` extras for whichever programming languages you use most frequently. I wouldn't install any other extras until you've either encountered them later in this book or had a chance to research them after you finish the book. Otherwise, they may change behaviours in ways that I won't have the foresight to write about.

You can find more information on each extra by visiting <https://lazyvim.org> and clicking the "Extras" menu item on the left menu bar. It includes links to the list of plugins each extra installs as well as the configuration LazyVim brings for that extra.

Disabling a Built-in Plugin

Sooner or later, you're going to want to edit your LazyVim configuration. The out-of-the-box defaults are wonderful, but the odds are that they don't 100% exactly match your personal needs (unless you are the LazyVim maintainer, in which case, I am compelled to say, "Hello, folke, I love LazyVim!").

While the vast majority of LazyVim's default plugins are no-brainers that you want to keep, you may find there are one or two plugins that you just don't need. In most cases, it doesn't actually matter, since LazyVim only loads plugins when you actually use them, so you can just ignore the ones that aren't relevant to you.

In my case, the one plugin I have disabled is `Neo-Tree`, as I foreshadowed in the previous chapter.

The LazyVim configuration can be opened from the dashboard by simply pressing the `c` key. Or you can use Space Mode to access the configuration files at any time using `<Space>fc` for "Find Config Files".

This will load the `lazyvim` config folder in Telescope. This folder is typically `$HOME/.config/nvim`. Neovim loads `$HOME/.config/nvim/init.lua` by default, and if you weren't using LazyVim, this is where you would do all your configuration.

With LazyVim, `init.lua` just uses the `lua require` statement to include the LazyVim configuration infrastructure. You will normally not have to touch this file, instead following the "LazyVim way".

In addition to a barebones `init.lua`, LazyVim has put a few configuration files and a bit of folder structure in the configuration directory.

For now, the main thing we need to know is that any `lua` files inside the `lua/plugins` subdirectory will be automatically loaded by LazyVim, no matter what their name is. I have a number of different files in this folder for my custom configurations.

I call the one that holds my disabled plugins `disabled.lua`. The easiest way to create this file is to open one of the existing config files and use either `neo-tree` or `mini.files` to create a new file, as described in the previous chapter.

When I created my `disabled.lua` file in the `lua/plugins` directory, my intention was to collect all the LazyVim plugins I don't want in it. Turns out that's a really short list! The contents of this file is simply:

```
return {  
  { "nvim-neo-tree/neo-tree.nvim", enabled = false },  
}
```

If there are any other plugins that LazyVim enables by default that you don't want to use, just follow the same syntax. The first argument in the lua table is a string containing the github repo (with owner) you want to disable. The second argument is to set `enabled = false`. That's it!

Tip: You will inevitably forget the `return` statement at the beginning of a plugins file at some point. Now you know to watch out for it.

If you don't know the Lua language... honestly, don't worry about it. I've never formally studied it, but I've picked up enough by osmosis to easily maintain my Neovim configuration.

If you're less foolish than me, you might want to type `:help lua` and read the official Neovim docs on the topic. followed by `:help lua-guide-api`.

Modifying Keybindings (example)

Keybindings are one of the few things I don't love about working with LazyVim, although it's not strictly LazyVim's fault. I just never quite know where to define the damn keybindings.

There are basically three possible places to configure keybindings, depending on how any given plugin is configured:

- In `.config/nvim/lua/config/keymaps.lua`. This is where you configure or modify keybindings that are not specific to plugins, but rather modify core NeoVim functionality. But sometimes, the plugin is configured such that you need to set up the plugin keybindings in this file, too.
- In the `keys` field of the lua table (in Lua, a "table" is like a record or dict in many other dynamic languages) passed to a plugin. This is typically where you map global

normal mode keybindings to set up a plugin. This is what we will do with `mini.files`.

- In the `opts` (options) argument passed into a plugin's configuration. The format of the options for any one plugin are plugin specific, but many plugins prefer to set up keymaps on your behalf through options instead of having you do the mapping yourself. This is especially true if the keymaps define a different "mode" or only apply if the plugin is currently open or active. I'll give an example of this with `mini.files` as well.

To demonstrate, I want to "fix" the fact that `mini.files` doesn't have a "open in root" option. I like the "open in directory of current file" option, but I also want to be able to open in the root directory.

Since we've disabled `neo-tree`, I'm going to steal the `<Space>e` and `<Space>E` keybindings and reuse them for `mini.files`, then I'll remap the existing `<Space>fm` keybinding to open the root so I can access all three modes. You can, of course, choose different keybindings if they map better to your mental model or you are keeping `neo-tree` around.

Using `mini.files`, I created a new file named `extend-mini-files.lua` in my `.config/nvim/lua/config/plugins/` directory. As with the `disabled.lua` file, this file can be named anything so long as it's in the `plugins` directory. I have a habit of prefixing any configuration that I am using to *overwrite* the defaults provided by LazyVim with the word `extend`.

This makes it easy to distinguish it from non-LazyVim plugins I've installed when I'm listing the directory using `mini.files` or `Telescope`.

Inside this new file, I used this code:

```
local Util = require("lazyvim.util")

return {
  "echasnovski/mini.files",
  keys = {
    {
      "<leader>e",
      function()
        require("mini.files").open(vim.api.nvim_buf_get_name(0), true)
      end,
      desc = "Open mini.files (directory of current file)",
    },
  },
}
```

```

{
  "<leader>E",
  function()
    require("mini.files").open(vim.loop.cwd(), true)
  end,
  desc = "Open mini.files (cwd)",
},
{
  "<leader>fm",
  function()
    require("mini.files").open(Util.root(), true)
  end,
  desc = "Open mini.files (root)",
},
},
}

```

The first line imports the `lazyvim.util` module so we can use the handy `Util.root()` function it provides. I found this by reading through the default configuration for the Telescope find files and Neo-tree plugins, which is conveniently provided on the LazyVim website.

Then we return a Lua table, wrapped in curly braces. Lua tables can act as an array and a dictionary at the same time. In this case, the first element in the table is the string `"echasnovski/mini.files"`. It doesn't have a named key, so it's kind of like a "positional argument".

The second element in the table is more like a "named argument" in that it is indexed with the name `keys`, and the value is another Lua table. However, the second table acts more like an "array" of three values (three more separate lua tables) because it doesn't have named indices.

It is important to understand that the `keys` field is **merged** with the keys that are provided by the default LazyVim (extras) configuration for `mini.files`. If there are conflicts (such as with `<space>fm`), my values take precedence over the defaults.

This is a powerful feature of LazyVim that allows you to use hosted configuration provided by LazyVim but override it as needed. Older Neovim distros tended not to have this level flexibility.

To be clear, `keys` is a LazyVim concept (technically, it's actually part of the underlying `lazy.nvim` plugin manager). Any plugin configuration can have a `keys` array table, and

those keybindings will be merged with the default NeoVim keybindings, the LazyVim keybindings, your custom global keybindings, AND any other plugin keybindings.

Yes, that's a lot of potential for conflicts, which is why I'm so glad LazyVim has done most of the configuration for me!

Structure of a keys entry

Each item in the `keys` table is another Lua table with (in this case) three fields. The first two fields are positional and represent the keybinding name and the lua callback function that gets called whenever that keybinding is invoked. The third field is a named field, `desc` and provides a string description that will be shown in the Space mode menu.

The keybinding sequence in the first entry is using a standard syntax that comes from Vim. Recall that `<leader>` is an old Vim concept that allows you to configure which key is used as the prefix for custom keybindings. In LazyVim (and indeed, for most modern Neovim users), the leader is `<Space>`. Special keys are indicated to Vim's keybinding engine using angle brackets, so you will often see notations such as `<Space>`, `<Right>`, `<Left>` or `<BS>`.

After the `<leader>` string, we include any additional keys that need to be pressed. For the simple ones, we have `e` and `E` to replace the `Neo-tree` keybindings we disabled with new `mini.files` keybindings. The third one is a bit more complicated, as the `f` indicates that this action will be available under the `file/find` submenu in Space mode, and the `m` indicates which letter will be in this menu.

For the callbacks, we use Lua functions, which always start with `function` and end with `end`. These are anonymous (unnamed) functions, and they don't accept any parameters inside the parentheses. The function bodies we call specific code to open `mini.files` the way we want. In two cases I just copied this code from LazyVim's default `mini.files` configuration, and in the third, I cobbled it together by combining code from the `Neo-tree` and `mini.files` configurations.

Customizing the mini.files Options

As I mentioned, the `keys` table is merged with the default `keys` table that LazyVim has configured for `mini.files`. Similarly, most NeoVim plugins can be configured with an `opts` table that contains custom configuration specific to that plugin. If you supply an `opts` table, it will be *merged* with the default LazyVim one (if there is one).

You'll need to read each plugin's documentation (often available on Github, and usually available with `:help plugin-name`) to know exactly what options are available for each plugin. You'll also need to review the default configuration that LazyVim is setting up for that plugin so you understand how it will merge.

In my case, I pass the following `opts` array to `mini.files`:

```
return {
  "echasnovski/mini.files",
  keys = {
    -- the keybindings from above
  },
  opts = {
    mappings = {
      go_in = "<Right>",
      go_out = "<Left>",
    },
    windows = {
      width_nofocus = 20,
      width_focus = 50,
      width_preview = 100,
    },
    options = {
      use_as_default_explorer = true,
    },
  },
}
```

The mappings table in `mini.files` is used to override the default keymappings that are active *while* the `mini.files` view is open. This is different from the *global* keymaps we defined earlier to open `mini.files`. In my case, I have mapped `go_in` and `go_out` to use the arrow keys instead of `h` and `l` because of the left-handed Dvorak Kinesis weirdness I described previously. I don't recommend you make this change; `h` and `l` will work better for most anybody who isn't me.

The window options are there because I have a 32" 6k monitor, which means I can afford to have larger-than-normal explorer columns. Refer to the `mini.files` help for more information on these options.

So now you know a little bit about configuring plugins in LazyVim. It is both a little bit easier and a little bit harder than configuring plugins from scratch:

- It is easier because you only need to change the values that are non-default, instead of setting up an entire configuration, and LazyVim comes with very sane defaults.
- But it is harder because you sometimes have to think about how the option and keybinding merging happens, which wouldn't be necessary if you just had one great

big configuration object to begin with. This merging can get quite tricky for plugins that have complicated default LazyVim configurations.

Modifying Existing Options

Sometimes the “merging” behaviour LazyVim uses to overwrite options with the ones you provide in your plugin overrides is too simplistic. This most often happens when you are modifying a plugin that calls or defines a function for options behaviour instead of customizing it.

To support this situation, the `opts` entry in a `lazy.nvim` plugin’s configuration table can be a function instead of a static table. The function accepts one argument, which is the previous `opts` table as it was configured by LazyVim. Your function needs to *modify* this table to suit your desired behaviour. Note that it **does not** return a new `opts` table; it needs to modify the one that was passed in.

For example, the default LazyVim configuration for the `nvim-cmp` plugin is a pretty long and complicated function. The `nvim-cmp` plugin is responsible for the completion pop-up menu that provides suggestions as you type. It is an insanely useful feature, but I don’t like that by default (in LazyVim), selecting a completion is done with an `Enter` keypress. This drives me nuts when editing text as I am right now because I press `enter` for new lines all the time, often ignoring the pop-up.

There are a couple recipes for modifying this behaviour in the LazyVim docs, and other recipes you can try in the `nvim-cmp` README. In my case, I’ve configured it as follows:

```
return {
  {
    "hrsh7th/nvim-cmp",
    ---@param opts cmp.ConfigSchema
    opts = function(_, opts)
      local cmp = require("cmp")

      opts.mapping = vim.tbl_extend("force", opts.mapping, {
        ["<Right>"] = cmp.mapping.confirm({ select = true }),
        ["<CR>"] = function(fallback)
          cmp.abort()
          fallback()
        end,
      })
    end,
  }
end,
```



```
    },  
  }  
}
```

This `opts` function accepts the LazyVim-defined `opts` table as its second parameter. My code changes those `opts` using the `tbl_extend` function provided by Neovim. I add a new ["<Right>"] key to accept the suggestion (this matches fish shell behaviour) and overwrite the <CR> key with abort completion behavior.

This is harder to maintain than if I just had the whole configuration the way I wanted it in the first place. But I am willing to accept that tradeoff for all the places that LazyVim configures things better than I would have done on my own.

Installing third-party plugins

Installing a third-party plugin is little different from configuring a Lazy-Vim provided plugin, except that you don't have to worry about how the keys and `opts` are merged with a default config.

Simply create a new Lua file in the `plugins` directory (named appropriately for the plugin). Inside the file, return a Lua table where the first entry is the GitHub repo and name of the plugin, with other configuration (`opts` and `keys`, among others) after that name.

For example, I like the `guess-indent.nvim` plugin to set my shift width based on the contents of the file I am currently editing. It is maintained by the github user `nmac427`, so my `plugins/guess-indent.lua` file looks like this:

```
return {  
  "nmac427/guess-indent.nvim",  
  opts = { auto_cmd = true, override_editorconfig = true },  
}
```

The `opts` table depends entirely on what the plugin expects. In this case, I read the `guess-indent.nvim` README and found two options that I wanted to set.

Most modern Lua plugins will be documented as having to call a `setup` function with a Lua table containing the configuration. If the plugin you are trying to set up does not have explicit Lazy.nvim instructions, don't worry! Whatever gets passed into that `setup` function is what you need to include in the `opts` passed to the LazyVim plugin manager.

Another third-party plugin I recommend is `chrisgrieser/nvim-spider`. I have a file named `nvim-spider.lua` in my `plugins` directory as follows:

```
return {  
  "chrisgrieser/nvim-spider",
```

```

keys = {
  {
    "w",
    "<cmd>lua require('spider').motion('w')<CR>",
    mode = { "n", "o", "x" },
    desc = "Move to end of word",
  },
  {
    "e",
    "<cmd>lua require('spider').motion('e')<CR>",
    mode = { "n", "o", "x" },
    desc = "Move to start of next word",
  },
  {
    "b",
    "<cmd>lua require('spider').motion('b')<CR>",
    mode = { "n", "o", "x" },
    desc = "Move to start of previous word",
  },
},
}

```

This plugin doesn't automatically set up keybindings, so I pass a `keys = table` to the plugin configuration. This array is **not** passed to the plugin. Rather, the keys are parsed by the `lazy.nvim` plugin manager and added to the global keybindings. It is convenient to keep the keys with the plugin so all the configuration is in one place.

I am satisfied with the default options that `nvim-spider` passes to its `setup` function (after reading the README), so I don't have to pass an `opts` array.

The best resource for finding third-party plugins is the github repository [rockerBOO/awesome-neovim](https://github.com/rockerBOO/awesome-neovim). The list is well-maintained and (most importantly) pruned regularly, so there are no outdated or unmaintained plugins on the list.

In practice, LazyVim already ships with the best-in-class versions of most plugins (built-in or as extras), so you won't have to add many of them, but if you come across any "I wish LazyVim could..." scenarios, the answer is probably "it does and the plugin to do it is listed in the Awesome Neovim repo".

Summary

In this chapter, we learned a little bit about how LazyVim integrates with the wider NeoVim plugin ecosystem. It provides sane default plugins and configuration, but makes it easy to customize that configuration for your own needs.

We learned that built-in, extras, and unknown third party plugins are all treated slightly differently (though consistently), and saw examples of how to install some of the plugins I personally find indispensable.

Now that you know how to open files and configure plugins, we can get back to some of the nuts and bolts of modal editing. You already know how to switch between Normal and Insert mode and you can navigate around your code. In the next chapter, we'll cover some basic editing features that blur the line between navigating and inserting text.

Basic Editing

Armed with the navigation keybindings you've already learned and the ability to enter and leave insert mode at will, your Vim editing experience is getting pretty close to on par with what you might be used to in non-modal editors.

However, moving around and inserting text is a very small part of the life of a software developer. More often, you need to *edit* text. Deleting code, changing code, refactoring code moving code around. It's the majority of what we do.

Yes, you can do all of these things by navigating to where you want to, and entering insert mode. The delete and backspace keys do the same thing in insert mode that they do in other editors. But there are far more efficient tools.

The best part is that you already know most of what you need to take advantage of very powerful editing commands!

The Vim Command Mental Model

The navigation commands such as `s` and `f` and `hjk1` and `w` that you already know are collectively known as *motion* commands. They move the cursor from its current location to a new location.

Most motion commands can be prefixed with a count, so the navigation model is always `<count><motion>`. The effect of a count is usually to repeat the motion a certain number of times, although some commands such as `Shift-G` for "Go to line" will use the count as an absolute value instead. If the count is blank, the "default" count is typically 1. Even a Seek

command which uses labels is allowed to be prefixed with a count (although the count will be ignored).

The `<count><motion>` commands are great for navigation, which is all we've used them for so far, but they can also be combined with a *verb* to do something to the text between the cursor and the location the motion would move you to.

Verbs come first, so the structure is always `<verb><count><motion>`. Navigation is the "default" verb, so if you leave the verb blank (i.e. skip it), your cursor moves to the location indicated by the motion. We'll discuss several important verbs in this chapter.

But the model keeps growing! It turns out, verbs can also be counted. The syntax becomes `<count><verb><count><motion>`. I have never in my life used all four of those in one command, however. Typically you would either do `<count><verb><motion>` OR `<verb><count><motion>`.

This is starting to look like a full fledged grammar (spoiler alert: it is).

This model is nice because it allows you to divide and conquer your learning strategy, and reuse knowledge as you learn more. First you learned motion commands. Then you learned counts. Now you will learn verbs. If you learn new motion commands or new verbs in the future, you can mix them with all the verbs and motions you already know and they should behave in a predictable way.

Various plugins try to mimic this strategy, and, well, most are successful. My main complaint with `Neo-tree` is that it doesn't operate with the `<verb><motion>` mental model, but `Mini.files` does. Similarly, some folks argue that Seek mode violates the vim mental model because counts don't make sense. My opinion is that Seek mode simply transcends counts, but it still combines cleanly with verbs so it is a valid vim model.

A Note on Insert Mode

Like all models, this one is not perfect. For example, you can use counts with the `i`, `I`, `a`, and `A` commands, but it's clear that "enter insert mode" is neither a motion nor a verb.

For example, if you type `5ifoo<Escape>`, Neovim will insert `foofoofoofoofoo` for you. That may not seem very useful, but if you ever want an 80 character * ruler to underline a heading, `80i*<Escape>` is pretty nifty!

But the `<count>i` "not-motion" commands *cannot* be combined with verbs like the navigation commands you've learned, so it's important to know the limits of the model.

So now that you understand how the motions you already know can combine with verbs to perform actions other than navigation, you just need to learn some verbs.

Deleting Text

I've previewed this a couple times already, and even if I hadn't, you can probably guess that the verb for deleting text is `d`.

So where `motion` will take you to a specific location in the code, `d<motion>` will delete all the text between the cursor and that location. Here are some examples:

- `dh` to delete the character to the left of the cursor.
- `d3w` to delete three words.
- `3dw` to delete one word, three times.
- `d^` to delete from the cursor to the beginning of the line.
- `d2fe` to delete all text between the cursor location and the second `e` after the cursor, including that second `e`.
- `d2Ta` to delete all text between the cursor and the second `a` *behind* the cursor, *not* including that second `a`.
- `dsfoo` to delete text between the current cursor position and the label `s` that pops up when you use Seek mode to seek to `foo`. Note that Seek mode **always** jumps to the beginning of the word you searched for. This means that if the `foo` you jump to is after the current cursor location, the `oo` will not be deleted, but the `f` will. But if the `foo` you jump to is before the current cursor location, all three letters of `foo` will be deleted.

If any of those are surprising, ignore the `d` and refer back to earlier chapters to refresh your memory of the motions.

So `d` will work with all the motion commands you know, as well as all the motion commands you don't yet, **and** all the motion commands that are defined by plugins you haven't yet installed.

When the delete command is completed, Neovim will still be in Normal mode, and you can immediately perform any other `<verb><motion><pair>` combination.

Changing Text

Sometimes you just want to delete text, but another common task is editing text. Replace a word with another word, change spelling (coincidentally, I just misspelled "change"), delete the rest of the paragraph and replace it with something new, etc.

This can easily be handled by combining delete and insert mode (e.g. `dwi` will delete a word and enter insert mode.) However, you can save a keystroke by using the `c` verb, which

means “change”. If you replace the `d` in each of the examples I outlined above with a `c`, you will effectively get “delete the text and immediately enter insert mode.”

Operating to end of the current line

It is very common to want to delete or change from the cursor position to the end of the current line, leaving the beginning of the line intact. These actions happen more often than you would expect in source code editing, so there is a shortcut for them.

Yes you could `d$` and `c$` to delete or change to the end of the line, since `$` is the “jump to end of line” motion. That is the “correct” format for the mental model. However, because this is such a common operation, you can “cheat” with one fewer keystrokes and just use `Shift-D` or `Shift-C` instead.

Note that there is no inverse shortcut verb for “delete to the beginning of the line”, so you’ll have to use `d^` or `d0` instead, where `^` is the motion to jump to the first non-blank character and `0` is the motion to jump to the first column.

Operating on entire lines

Another common action is to change or delete an *entire* line of text. So much so, in fact, that there are special motions for “the whole line”. These motions are accessed by duplicating the verb. This is another place where the mental model kind of breaks down; the interpretation of the motion *depends* on the verb.

In practice, this just means that `dd` deletes an entire line and `cc` deletes it and enters insert mode. These are nice and easy to type, so it makes for a nice shorthand.

You can combine these bespoke motions with counts. `d3d` will delete three lines, and `3dd` will delete one line three times (which is faster to type because you don’t have to move your finger off of `d` to hit it twice). Yes, that has the same outcome either way, but the model is such that you can use either of them. Note that there are situations where the two formats may have subtly different behaviours, although in practice I have never encountered surprises.

Some shortcuts for modifying individual characters

Another common operation is to perform a delete or change operation on a single character or specific number of characters. You could do this using `d1` to delete the character under the cursor or `4d1` to delete that character and the three characters that come after it. However, because you do this so often, there is a shorthand verb that doesn’t have a motion (or rather, the motion is implied): `x`. For example, you can use `x` to delete an extraneous `u` in words like `behaviour` if your editor is from the USA but you live in a member of the

Commonwealth. The single letter will be deleted, and you'll be back in normal mode ready to proceed.

The `x` command can be used with a count, so if you want to delete five characters starting with the one under the cursor, just use `5x`.

If you need to go the other direction and delete characters *before* the cursor, use `Shift-x`. This, too, can have a count, and it will basically delete that many characters to the left. I rarely use this, since the shift brings us up to two keystrokes anyway, and `hx` or `d4h` is no harder.

If, instead of deleting, you need to replace a character with a different character, use the `r` command. This command will briefly enter insert mode while you type one character, then immediately return to normal mode. Much fewer keystrokes for a common operation (spelling errors are common, right? It's not just me?) than something like `cle<Escape>`. Using `r` with a count is possible, but the behaviour is kind of unhelpful: it will replace the character under the cursor and the appropriate number of characters after that character with the same letter. The only place I can imagine this being helpful is when you copy-paste a password prompt from somewhere and need to replace all the characters in the password with `*`.

Another common operation is deleting the newline at the end of the current line. Use the `Shift-J` (`j` stands for "Join Lines") command from anywhere in the line. I use this one a lot. If you need to merge multiple consecutive lines together, `Shift-J` takes a count. It generally does the right thing around whitespace (replacing indentation with a single space), but if you need to do a join without modifying whitespace, use the two-character combination `gJ`.

Manipulating Case

If you need to convert a character or sequence of characters to uppercase, use the verb `gU` (That's a `Shift-U` for the second character) followed by any standard navigation motion. (Nobody said a verb had to be a single letter, though most are). I find this particular verb frustrating because `g` is normally assigned to the `Go To` motions. In this case, (as with `gJ` above) it is a verb instead.

I guess you can think of it as "Go To and Convert to Uppercase" where `U` is short for Uppercase.

The inverse function to convert all text between the current cursor position and the motion destination is to use a lowercase `gu` before the motion. Kind of weird to remember, but it does match the common vim idiom of "u" means an action and "U" means the same action BUT BIGGER.

The duplicate commands `gUgU` and `gugU` do the same thing as other duplicate verbs, applying the upper/lower case operation to the entire line. It's a rather annoying sequence of keypresses, though, little easier than combining `gU` with the `^` and `$` motions (i.e. `^gU$`).

I don't find these commands very useful. I more frequently use the `~` command, which inverts the case of the character under the cursor.

Tip: If you find yourself doing a lot of case switching work, have a look at the [coerce.nvim](#). It doesn't have a LazyVim extra so you'll need to configure it yourself, but it can be worth the effort.

Repeating Commands

LazyVim doesn't have multiple cursor mode. There *are* plugins to support multiple cursors, but in my experience they don't work very well. Neovim does have multiple cursors on their roadmap, so I am hoping they will come up with a paradigm that integrates nicely with the vim mental model.

In the meantime, Neovim provides several different tools available for performing an action in multiple places in your code. We'll cover basic repetitions here, and other useful techniques in later chapters.

Once you have performed any verb, you can navigate to another place in the document and repeat that verb with a single keypress: `.` (That's a period, although you will usually hear it referred to as "dot repeat" in this context).

This highlights why `d` and `c` need to be separate verbs, as opposed to using something like `d<motion>i`. When you use `c`, the delete motion **and** the text you inserted is remembered, so you can repeat the entire change with a `.` command. For example, if you want to replace all instances of a variable named `i` with a much better name of `index`, you could jump to the first instance of `i` and type `c1index<Escape>` to "change one character to index". Then you can use Seek mode or other navigation commands to go to the next use of `i`. Now just type `.` to repeat the change and continue to the next instance.

Like motions and verbs, the `.` command can be given a count. However, counts with `.` are a little bit nuanced. Rather than blindly repeating the command `<count>` number of times, it will instead *replace* the count of the command being repeated.

This means that if you use the verb `3dd` to delete three lines, and the next operation you perform is `2.` ("2 dot"), the second operation will delete *two* lines, rather than six.

Recording Commands

Vim's command recording and playback system is extremely powerful. You can trivially record an arbitrary sequence of navigation, editing, and insertion commands, then repeat that sequence on demand at any location you want.

To start a recording, press `qq`. Sorry, but I have no mnemonic to remember `q`. I have a feeling it was just the last available key on the keyboard!

After that, type whatever sequence of navigation, editing, and insertion commands you want to record. Delete words, insert text, change text, search for words (don't use Seek mode, as the replay mechanism will have no idea what label to jump to). Virtually anything you can do in vim (even `:` commands) can be recorded and replayed later.

When you are finished recording, just press `q` again. The recording will be stored ready for replay whenever you desire.

Appending to a recording

If you partially complete your recording and then realize you need some more information or need to make an edit before completing the recording, you can stop the recording using `q` as usual and do the thing you need to do.

When you are ready to continue recording, use `qQ` to record in *append* mode instead. The main tip here is that you need to make sure your cursor is in a location such that the merged recording will make sense. This usually means the same place it was when you stopped recording, although it may depend on what changes you made in the meantime.

Playing Back a Recording

The easiest and fastest way to play back your most recently saved recording is with `Shift-Q`.

It is possible to store and replace multiple recordings at once using *registers* (a stupid name for a storage location that harkens back to humanity's dark days of assembly programming). I will go into more detail about registers in a later chapter.

Undo and Redo

Obviously, these are the most important operations in the whole book! Use the `u` key to undo your most recent change. Note that "most recent change" can be a pretty big whack of text, especially if you haven't exited Insert mode for a while. For example, I wrote this entire paragraph in one Insert session. If I press `u` the entire paragraph will be lost.

That's ok, though, because I can redo using `Control-r`. Like most developers, I use both of these extensively. (Did you know that in the old days of typewriters, secretaries had to get 100% accuracy scores on their typing tests? There was no backspace or delete key, you see).

Neovim actually does an amazing job of keeping track of your entire history, rather than just the most recent suite of changes. So if you make a bunch of changes to get to state B, then undo to state A, and then make a bunch more changes to get to state C, it is still possible to get back to state B (ie: back out of the C changes to state A and go back up the B changes to state B).

It's kind of the same concept as git branches, except your history is automatically tracked for every keystroke you make. Working with branches of undo history using raw Neovim commands can feel pretty clumsy, though (read through `:help undo-branches` if you're brave). Instead I recommend configuring and installing the [undotree](#) plugin.

About 99.9% of the time, `u` and `Control-r` will be all you need, but that remaining 0.1% can be a godsend when you need it.

Summary

In this chapter, we expanded our understanding of the Vim mental model, and then introduced several verbs that can be combined with the navigation motions we were already familiar with.

We discussed a grab-bag of other editing commands before covering how to repeat motions using `.` and command recordings. Finally, we covered undo and redo.

In the next chapter, we'll learn about text objects and some additional nuances of the vim mental model with operator-pending mode. Combined, these allow us to very quickly perform actions on a huge variety of code concepts.