2ND EDITION

<packt>

MASTERING

Efficient and effortless editing with Vim and Vimscript

RUSLAN OSIPOV

Reviewed by Bram Moolenaar and Christian Brabandt

Mastering Vim

Efficient and effortless editing with Vim and Vimscript

Ruslan Osipov

‹packt›

Mastering Vim

Associate Group Product Manager: Kunal Sawant
Publishing Product Manager: Samriddhi Murarka
Senior Content Development Editor: Rosal Colaco
Book Project Manager: Deeksha Thakkar
Technical Editor: Jubit Pincy
Copy Editor: Safis Editing
Indexer: Tejal Soni
Production Designer: Gokul Raj S.T
DevRel Marketing Coordinator: Shrinidhi Manoharan
Business Development Executive: Debadrita Chatterjee

First published: November 2018
Second edition: July 2024

Production reference:1190724

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-83508-187-7

www.packtpub.com

Dedicated to my mother, my grandmother, and my wife, the three most important women in my life.

– Ruslan Osipov

Contributors

About the author

Ruslan Osipov is a software engineering manager at Google, and the author of the bestselling "Mastering Vim" (first edition, 2018). Passionate about developer productivity and workflow optimization, he continues to refine his Vim expertise and share his knowledge with the community in this expanded second edition.

I'd like to say thank you to Samriddhi, Deeksha, Rosal, and everyone else from Packt who worked on this book. A huge thank you to Bram for reviewing the first edition of this book (and may he rest in peace), and Christian. A word goes out to the Vim Japan conference organizers—Tatsuhiro Ujihisa (Uji), Taro Muraoka (KaoriYa), Thinca, Aomoriringo, Mopp, Yasuhiro Matsumoto (Mattn), t9md, and Guyon (and anyone else I missed). Thank you for your hospitality!

Special thank you goes to Masafumi Okura, who diligently identified a number of inaccuracies in the book when translating it to Japanese.

About the reviewers

Bram Moolenaar (first edition, 2018) was the creator and maintainer of Vim. With the help of volunteers, he worked on it for 32 years. Bram passed away in 2023.

After studying electronics and inventing parts of digital copying machines, Bram decided that creating open source software was more useful and fun, so he worked on that exclusively for several years. He last found himself employed at Google, one of the few companies that fully embrace open source software. In between, he did voluntary work on a project in Uganda and is still helping poor children there through the I Care Children Foundation (ICCF).

"I would like to thank all the Vim developers for helping me make Vim into what it is today. Without them, only a fraction of the features would have been implemented and the quality would not have been nearly as high. I would also like to thank all the plugin writers for building on top of Vim and making complex features available to users (so that I don't have to!). And finally, I would like to thank Ruslan for writing a book that not only aids users with Vim's built-in features, but also with getting to know and use plugins."

Bram Moolenaar, 2018

Christian Brabandt (second edition, 2024), has been working in the IT industry for almost 25 years, mostly as a consultant in various roles, and enjoys working with people and helping them implement a complex technology stack. He is currently employed with Ataccama as Senior Platform Consultant where he often has to work with Vim in various incarnations and on different platforms.

He has been involved within the Vim community in various roles ever since graduating from university and slowly taking a bigger role in Vim development. Since 2023 he has been one of the main maintainers of the Vim project and helped lead the project after the passing of Bram Moolenaar, the long-standing Vim maintainer.

I'd like to thank my family and friends for helping me through some personal dark times, allowing me to take away their precious time, spending it on Vim development, and investing it into the Vim community. Thank you all for your support and for allowing me to be who I am!

I'd also like to thank everybody in the Vim community for spreading the word and contributing to Vim in different ways and keeping the project healthy!

Table of Contents

Preface

Mastering Vim will introduce you to the wonderful world of Vim through examples of working with Python code and tools in a project-based fashion. This book will prompt you to make Vim your primary IDE since you will learn to use it for any programming language.

Who this book is for

Mastering Vim is written for beginner, intermediate, and expert developers. The book will teach you to effectively embed Vim in your daily workflow. No prior experience with Python or Vim is required.

What this book covers

Chapter 1, *Getting Started*, introduces the reader to basic concepts and the world of Vim.

Chapter 2, *Advanced Editing and Navigation*, covers movement and more complex editing operations and introduces many plugins.

Chapter 3, *Follow the Leader – Plugin Management*, talks about modes, mappings, and managing your plugins.

Chapter 4, *Understanding the Text*, helps you interact with, and navigate, code bases in a semantically meaningful way.

Chapter 5, *Build, Test, and Execute,* explores options for running code in, or alongside, your editor.

Chapter 6, *Refactoring Code with Regex and Macros*, takes a deeper look at refactoring operations.

Chapter 7, *Making Vim Your Own*, discusses options available for further customizing your Vim experience.

Chapter 8, *Transcending the Mundane with Vimscript*, dives into the powerful scripting language Vim provides.

Chapter 9, *Where to Go from Here*, provides some farewell food for thought, talks about Vim's younger sibling, and points at a few places on the internet you might be interested in.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Vim-Second-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Functions in Vimscript 9 are defined using the `def` keyword."

Keypresses are indicated as follows: *jk*. This means keypress *j*, followed by keypress *k*. More complex keypress chords are written out explicitly (e.g. *Ctrl* + *j*, *k*).

A block of code is set as follows:

```
" Manage plugins with vim-plug.
call plug#begin()
call plug#end()
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
" Manage plugins with vim-plug.
call plug#begin()

Plug 'scrooloose/nerdtree'
Plug 'tpope/vim-vinegar'
Plug 'ctrlpvim/ctrlp.vim'
Plug 'mileszs/ack.vim'
Plug 'easymotion/vim-easymotion'

call plug#end()
```

Any command-line input is written as follows:

```
$ cd ~/.vim
$ git init
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

> **Warning**
> Warnings or important notes appear like this.

Tip
Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Mastering Vim*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-83508-187-7`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1
Getting Started

Welcome to Mastering Vim, a book that will teach you how to get good with Vim, its plugins, and its ideological successors!

This chapter will establish a foundation for working with Vim. Every tool is built with a particular usage philosophy in mind, and Vim is no exception. Vim introduces a different way of working with text compared to what most people are used to these days. This chapter focuses on highlighting these differences and establishing a set of healthy editing habits. It will let you approach Vim with a Vim-friendly frame of mind and will ensure you're using the right tools for the job. To make examples concrete, we will be using Vim to create a small Python application throughout this chapter.

The following topics will be covered in this chapter:

- The difference between major Vim versions
- Modal versus modeless interfaces, and why is Vim different from other editors
- Installing and updating Vim
- The gVim – the graphical user interface for Vim
- Configuring Vim for working with Python and editing your configuration
- Common file operations – opening, modifying, saving, and closing files
- Moving around – navigating with arrow keys and cursor movement keys, by words, paragraphs, and so on
- Making simple edits to files and combining editing commands with movement commands
- Persistent undo history
- Navigating the built-in Vim manual

Technical requirements

Throughout this chapter, we will be writing a basic Python application. You don't have to download any code to follow along with this chapter as we'll be creating files from scratch. However, if you ever get lost and need more guidance, you can view the resulting code on GitHub:

`https://github.com/PacktPublishing/Mastering-Vim-Second-Edition/tree/main/Chapter01`

We will be using Vim to primarily write Python code throughout this book, and it is assumed that the reader is somewhat familiar with the language. Examples assume you're using **Python 3** syntax.

> **Working with Python 2 code**
>
> If you must live in the past, you can convert Python 3 examples to **Python 2** code by changing the `print()` command syntax. Change all of `print('Woof!')` to `print 'Woof!'` to make the code run in Python 2.

We will also be creating and modifying Vim configurations, which are stored in a `.vimrc` file. The resulting `.vimrc` file is available from the previously mentioned GitHub link.

A brief history lesson

Let's go back to the beginning of time: the middle of the twentieth century. Before personal computers and terminals, there were teleprinters. A teleprinter is a mechanical typewriter that can send and receive messages over a telecommunications channel. Here's a photo of the Teletype ASR-33 (1963) teleprinter – a fancy typewriter that was used as an interface and an input device for a computing machine:

Figure 1.1 – Teletype Corporation ASR-33 teleprinter (image by Arnold Reinhold, Wikipedia (CC BY-SA 3.0))

The only way to engage with the text through teleprinters was line by line, so line editors such as **ed** (developed by Ken Thompson) or its successor – Bill Joy's **ex** – were used. Line editors were, just like Vim is today, modal editors (more about that in a bit) but, due to aforementioned input/output limitations, were limited to working on one line at a time. Yes, really.

As technology progressed, teleprinters were replaced by terminals with screens, which enabled much more robust text editing. Vim's direct predecessor, **vi**, started its life all the way back in 1976. Unlike its predecessors, **vi** (developed by Bill Joy, the author of **ex**) included many quality-of-life features and even allowed you to edit multiple lines of text at once – what a luxury!

> **Did you know?**
>
> On many modern systems, the `vi` command is a symlink to a feature-limited version of `vim` referred to as `vim-tiny` (rather than the original vi implementation)!

Vi inspired many clones, including **STEVIE** (**ST Editor** for **VI Enthusiasts**): and **STEVIE** source code is what was eventually used as a basis for **Vim**. The first version of **Vim** was released in 1991 by Bram Moolenaar. Now, you're all caught up!

While it's not very likely that you'll stumble upon a device with Vim that hasn't been updated since the late 1990s, it helps to get a basic idea of what changed between different Vim versions.

Here, you can see the (oversimplified) highlights of each major version of Vim up until the moment of writing the second edition of this book. Vim has been in some form of continuous development since 1991!

Major version	Years	Highlights
1.0	1991	Bram Moolenaar releases **Vi Imitation** for the Amiga computer.
2.0	1993	**Vi Improved** is released under its modern name!
3.0	1994	Multiple windows.
4.0	1996	Graphical interface support.
5.0 – 5.8	1998 – 2001	Syntax highlighting, scripting, and select mode are added.
6.0 – 6.4	2001 – 2005	Plugin support and folding support are added.
7.0 – 7.4	2006 – 2013	Notable new features include spell checking, code completion, tabs, branching history, and persistent undo.
8.0 – 8.2	2016 – 2019	Optimizations, asynchronous I/O support, built-in terminal, and pop-up windows are available.
9.0	2022	Introduction of the new scripting language (Vim9script).

Just like with **vi**, there are many Vim clones, and some are quite successful. This book covers one of the more successful alternative implementations of Vim in *Chapter 9*.

Ask for `:help`

If you're interested in a more detailed breakdown of the differences between the versions, you can open `:help` (you'll learn how to use that by the end of the chapter) and look for **"Versions"**. Try `:help version9` to see what's new in **Vim 9**!

Let's start a conversation (about modal interfaces)

If you've ever edited text before, you are most likely to be familiar with modeless interfaces. It's the default option chosen by modern mainstream text editors, and that's how many of us learned to work with text. If you're old enough to remember a time before smartphones, many landlines and early mobile phones were also modeless.

The term **modeless** refers to the fact that each interface element has only one function. Each button press results in a letter showing up on screen, or some other action being performed. Each key (or a combination of keys) always does the same thing: the application always operates in a single mode.

However, this is not the only way.

Welcome to the **modal** interface, where each trigger performs a different action based on context. The most common example of a modal interface that we encounter today is a smartphone. Each time we work in different applications or open different menus, a tap on the screen performs a different function.

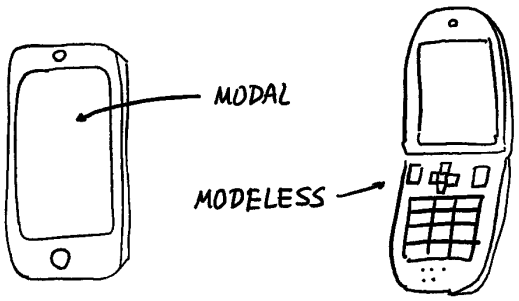

Figure 1.2 – A smartphone uses a modal interface, while the traditional phone is (mostly) modeless

It's similar when it comes to text editors. Vim is a modal editor, meaning that a single button press might result in different actions, depending on context. Are you in insert mode (a mode for entering text)? Then, hitting *o* would put the letter **o** on the screen. However, as soon as you switch to a different mode, *the letter o* will change its function to add a new line below the cursor.

Working with Vim is like having a conversation with your editor. You tell Vim to delete the next three words by pressing *d3w* (**d**elete **3** **w**ords), and you ask Vim to change the text inside quotes by pressing *ci"* (**c**hange **i**nside **"** [quotes]).

You may hear very frequently that Vim is faster than other editors, but it's not necessarily the point of Vim. Vim lets you stay in the flow when working with text. You don't have to break the pace to reach for your mouse, you don't have to hit a single key exactly 17 times to get to a particular spot on the page. You don't have to drag your mouse millimeter by millimeter to ensure you capture the right set of words to copy and paste.

When working with a modeless editor, workflow is filled with interruptions. Working with modal editors has a certain sense of flow to it: you ask the editor to perform actions in a consistent language. With Vim, editing becomes a much more deliberate exercise.

Installation

Vim is available on every platform, and comes installed on Linux and macOS (however, you may want to upgrade Vim to a more recent version). You have different options for setting up Vim depending on your operating system and preference, and here's a handy crude drawing showing some of the more common options:

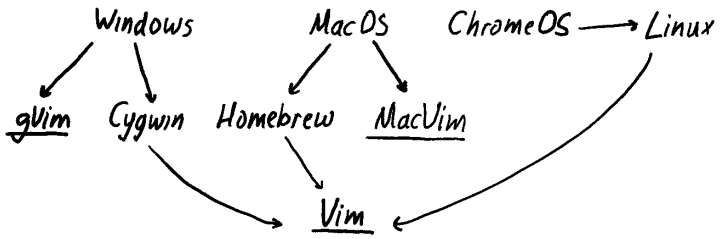

Figure 1.3 – Options for installing Vim across different OSs

Find your system in the following sections, and skim through the instructions to set it up.

Why so many screenshots?

You'll see that I've included a large number of details and screenshots in the following installation instructions. While Vim is easily available on most platforms, getting the latest version of Vim installed is not as straightforward as one would expect. If you realize that you're using the wrong version of Vim – you can always go back to the instructions in this chapter for help.

Setting up on Linux and Unix-like systems

Linux machines come with Vim installed, which is great news! However, it might be rather out of date, and a new version of Vim often includes new functionality and optimization changes (you can read more about changes between versions in the *A brief history lesson* section of this chapter). Pull

up your Command Prompt and run the following code to build an up-to-date Vim from the latest patch (at the time of cloning the source repository):

```
$ git clone https://github.com/vim/vim.git
$ cd vim/src
$ ./configure --prefix=/usr/local --with-features=huge
$ make
$ sudo make install
```

Keep on reading to learn more about *Compilation options* in the next section.

Missing dependencies

If you're running into issues as you're installing Vim, you might be missing some dependencies. If you're using a Debian-based distribution, the following command should add common missing dependencies:

```
$ sudo apt-get install make build-essential \
  libncurses5-dev libncursesw5-dev --fix-missing
```

This will make sure that you're on the latest major and minor patches of Vim. If you don't care about being on the cutting edge, you can also update Vim using a package manager of your choice. Different Linux distributions use different package managers; the following list includes some common ones:

Distribution/System	Command to install the latest version of Vim
Debian-based (Debian, Ubuntu, Mint)	`$ sudo apt-get update` `$ sudo apt-get install vim-gtk`
CentOS (and Fedora prior to Fedora 22)	`$ sudo yum check-update` `$ sudo yum install vim-enhanced`
Fedora 22+	`$ sudo dnf check-update` `$ sudo dnf install vim-enhanced`
Arch	`$ sudo pacman -Syu` `$ sudo pacman -S gvim`
FreeBSD	`$ sudo pkg update` `$ sudo pkg install vim`

Pay attention to names

You can see in the preceding table that Vim uses different package names for different repositories. Packages such as `vim-gtk` on Debian-based distributions or `vim-enhanced` on CentOS come with more features enabled (such as GUI support for instance).

Do keep in mind that package manager repositories tend to lag behind from anywhere between a few months to a few years.

That's it; you're now ready to dive into the world of Vim! You can start the editor by typing the following command:

```
$ vim
```

`vim` **versus** `vi`

Vi is Vim's predecessor (Vim stands for **Vi Improved**) and is available to be invoked via the `vi` command. On some distributions, the `vi` command links to a feature-stripped version of Vim (aka `vim-tiny`), while on some it's merely a symlink to a feature-complete Vim.

Compilation options

Compiling from source is often the best way to receive the latest available version of Vim. If you're not afraid to get your hands dirty, you might want to know about common compilation options.

This section will cover various options for the configure command. To compile Vim, you'll have to run the following commands, with `<options>` replaced with the desired compilation options:

```
$ git clone https://github.com/vim/vim.git
$ cd vim/src
$ ./configure <options>
$ make
$ sudo make install
```

You can control installation location with --prefix:

- `--prefix=/usr/local` as a reasonable default for a system-wide installation
- `--prefix=$HOME/.local` will make the newly installed Vim only available to your user

Feature sets allow you to enable different sets of features to be available in Vim, with options being tiny, small, normal, big, and huge. The most interesting options include the following:

- `--with-features=huge` includes all possible features, including experimental ones
- `--with-features=normal` includes a reasonable feature set, which is mostly covered in this book

- `--with-features=tiny` only offers bare-bones essentials, without syntax highlighting or plugin support

Language support is controlled via the `--enable-<language>interp` option. Note that this refers to the ability of Vim's internals to interact with the selected programming language (e.g., in plugins or your own scripts), and not your ability to edit said files. Some options include the following:

- `--enable-luainterp=yes` enables Lua support
- `--enable-perlinterp=yes` enables Perl support
- `--enable-python3interp=yes` enables Python 3 support
- `--enable-rubyinterp=yes` enables Ruby support

Finally, to integrate the clipboard with the X11 Window System (that is, your system-wide clipboard on Linux), you might want to compile Vim with the `--with-x` option. Note that you might need the X development library (e.g., you can use `sudo apt install libx11-dev libxtst-dev` if you're using the `apt` package manager).

Setting up on macOS

macOS comes prepackaged with Vim, but the version can be outdated. There are a few ways to install a fresh version of Vim, and this book will cover two. First, you can install Vim using Homebrew, a package manager for macOS. You'll have to install Homebrew first, though. Second, you can download a `.dmg` image of MacVim. This experience would be more familiar because Mac users are used to the visual interface.

Since this book covers interactions with the command line, I recommend taking the Homebrew route. However, you're welcome to go forward with installing the image if interacting with the command line does not interest you.

Using Homebrew

Homebrew is a third-party package manager for macOS, which makes it easy to install and keep packages up to date. Instructions on how to install Homebrew are available on `https://brew.sh`, and, as of the moment of writing this book, consist of a single line executed in the following command line:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/HEAD/install.sh)"
```

Once you run that prompt, hit *Enter* to continue (as many times as you need to).

If you don't have Xcode

If you don't have Xcode installed (which is often a prerequisite for any kind of development-related activity on Mac), you'll get an Xcode installation popup. We won't be using Xcode directly, and you can install it with default settings.

This should take a while to run, but you'll have Homebrew installed by the end: a fantastic tool you can use to install a lot more than Vim! You'll see the **Installation successful!** message in bold font once the installation is complete.

Let's install a new version of Vim now using the following command:

```
$ brew install vim
```

Homebrew will install all the necessary dependencies too, so you won't have to worry about a thing.

If you already have Homebrew installed, and you have installed Vim in the past, the preceding command will produce an error. You may want to make sure you have the latest version of Vim, though, so, run the following command:

```
$ brew upgrade vim
```

You should now be ready to enjoy Vim; let's try opening it with the following command:

```
$ vim
```

Welcome to Vim:

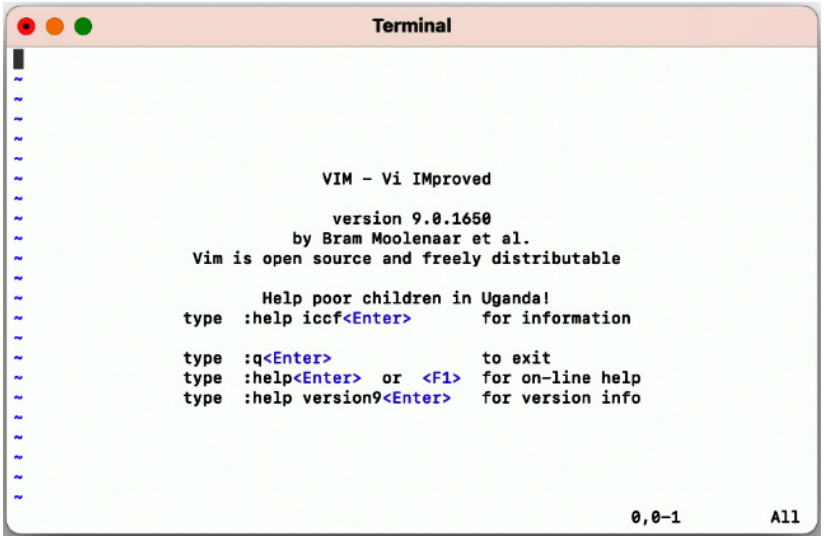

Figure 1.4 – Vim on macOS (installed via Homebrew)

Let's move on to the next section about downloading a `.dmg` image.

Downloading a .dmg image

Navigate to `https://github.com/macvim-dev/macvim/releases/latest` and download `MacVim.dmg`.

Open `MacVim.dmg`, and then drag the Vim icon into the `Applications` directory, as can be seen in the following screenshot:

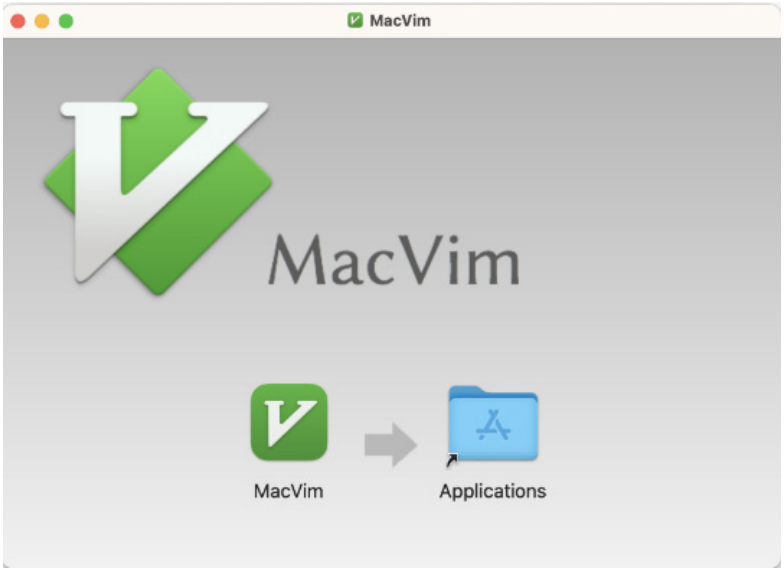

Figure 1.5 – The MacVim installation screen – drag and drop MacVim into the Applications folder

Depending on the security settings of your Mac, you might be greeted by an error when navigating to the `Applications` folder and trying to open the `MacVim` app, as demonstrated in the following screenshot:

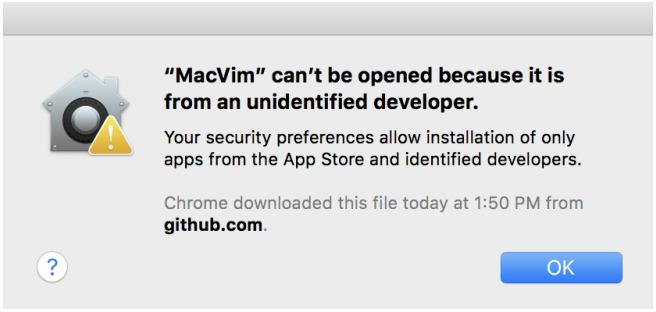

Figure 1.6 – The default "unidentified developer" prompt

Open your `Applications` folder, find `MacVim`, right-click the icon, and select **Open**. The following prompt will pop up:

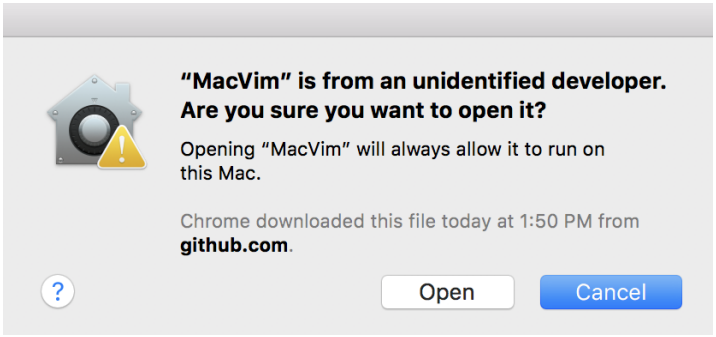

Figure 1.7 – The "unidentified developer" prompt, which you can get by right-clicking and selecting Open

Now, hit **Open**, and `MacVim` can be opened as usual from now on. Give it a shot:

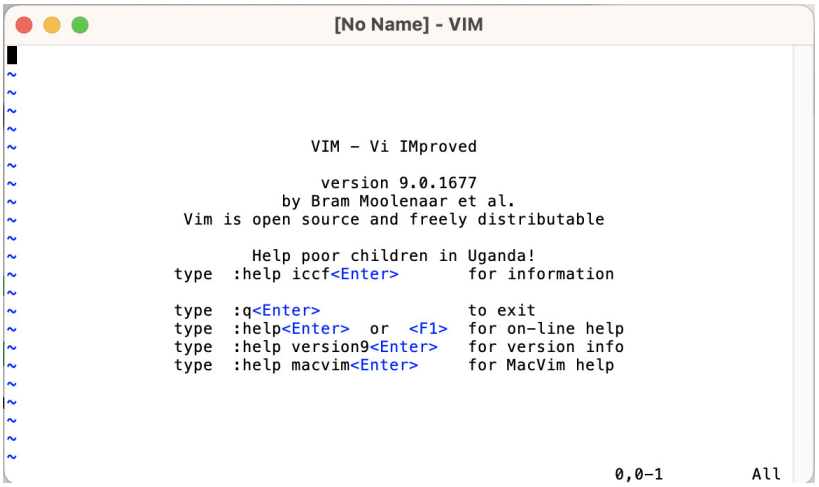

Figure 1.8 – MacVim on macOS

Setting up on Windows

Windows provides two primary routes for using Vim: setting up Cygwin and providing a more Unix-like command-line experience, or installing **gVim** – a **graphical version of Vim** (which supports working with `cmd.exe` on Windows). I recommend installing both and picking your favorite: gVim feels slightly more at home on Windows (and it is easier to install), while Cygwin might feel more at home if you're used to the Unix shell.

Unix-like experience with Cygwin

Cygwin is a Unix-like environment and a command-line interface for Windows. It aims to bring a powerful Unix shell and supporting tools to a Windows machine.

> **Windows Subsystem for Linux (WSL)**
>
> WSL is a feature in Windows 10+ that allows you to run a Linux environment directly in Windows. While I personally haven't had experience with WSL, it's widely praised as a fast, user-friendly, and reliable way to access the Linux command line and tools on Windows. It could be a better alternative to Cygwin as it continues to be developed. You can read more about WSL at `https://learn.microsoft.com/windows/wsl`.

Installing Cygwin

To begin the installation process, navigate to `https://cygwin.com/install.html` and download either `setup-x86_64.exe` or `setup-x86.exe`, depending on the version of Windows you're using (64-bit or 32-bit respectively).

> **How many bits are in your system?**
>
> If you're not sure whether your system is 32-bit or 64-bit, you can open **Control Panel | System and Security | System**, and look at **System type**. For example, my Windows machine shows **System type: 64-bit Operating System, x64-based processor**.

Open the executable file, and you will be greeted by the following Cygwin installation window:

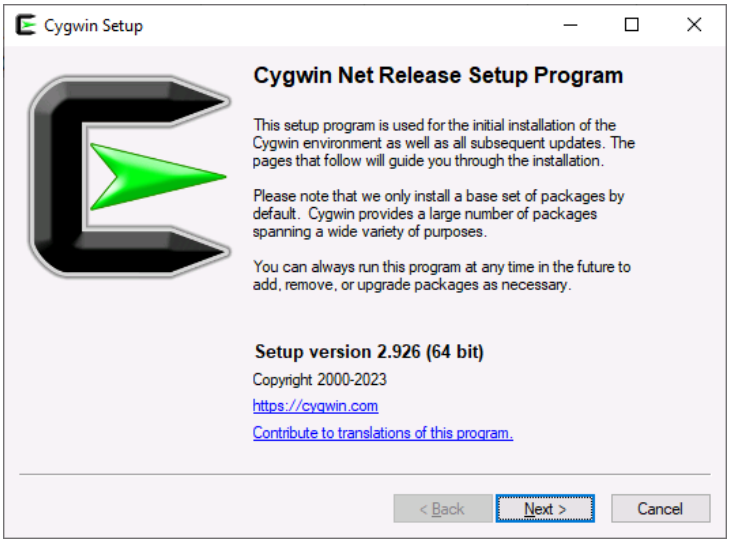

Figure 1.9 – The Cygwin Setup screen on Windows

Hit **Next >** a few times, proceeding with the default settings:

- **Download source**: **Install from Internet**
- **Root directory**: `C:\cygwin64` (or a recommended default)
- **Install for**: **all users**
- **Local package directory:** `C:\Downloads` (or a recommended default)
- **Internet connection: Use System Proxy Settings**
- **Download site:** `http://cygwin.mirror.constant.com` (or any available option)

After this, you will be greeted with the **Select Packages** screen. Here, we want to select the `vim`, `gvim`, and `vim-doc` packages. The easiest way to do this is to type `vim` in a search box, expand the **All |Editors** category, and click on the arrow-looking icons next to the desired packages, as demonstrated in the following screenshot:

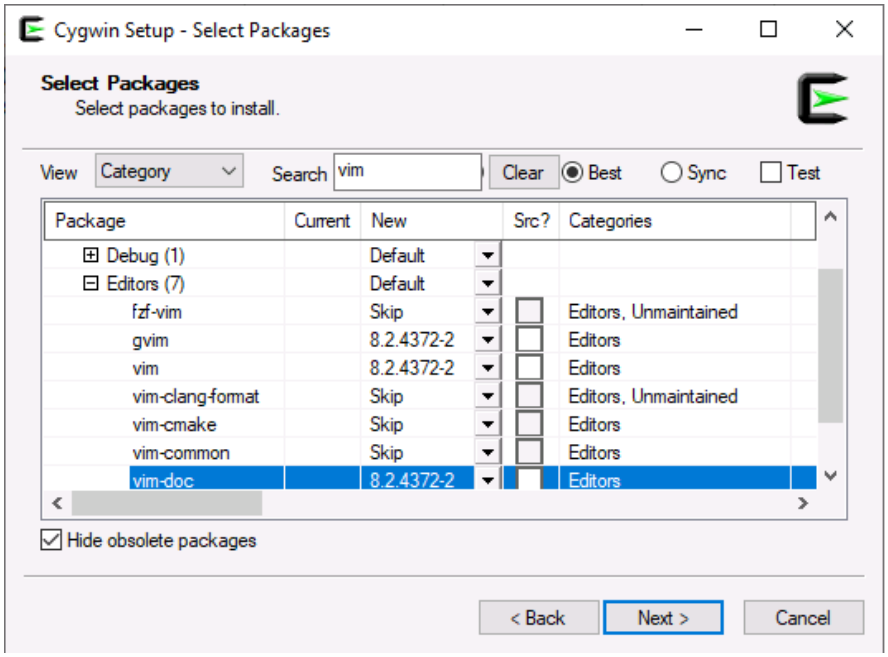

Figure 1.10 – Cygwin package selection screen – note that gvim, vim, and vim-doc are marked to be installed, as seen in the New column

The preceding screenshot shows version **8.2.4372-2**. This is the latest version available at the moment of writing this chapter, July 2023. At this time, the latest version of Vim is 9.0, which introduces **Vim9script** (you can learn more about version differences in the *A brief history lesson* section).

Use Cygwin to compile Vim

If you'd like to have the latest version of Vim, I recommend using Cygwin to compile Vim from its official Git repository. After installing Cygwin, you should visit the *Setting up on Linux* section we covered earlier in this chapter for the instructions. If you'd like to do that, you'll want to install `git` and `make` utilities in Cygwin.

You might need additional utilities

You may want to install `curl` from under the Net category, and `git` from under the Devel category, as we'll be using both in *Chapter 3*. It might also be helpful to install `dos2unix` from under the Utils category, which is a utility used for converting Windows-style line endings to Linux-style line endings (something you might run into once in a while).

Hit **Next >** two more times to proceed, which will begin the installation. The installation will take some time, and now would be a great moment to prematurely congratulate yourself with some coffee!

You might get a few post-install script errors, which you can safely dismiss (unless you see any errors related to Vim – then, Google is your friend: search for an error text and try to find a solution).

Hit **Next >** a few more times, proceeding with the defaults:

- **Create icon on Desktop**
- **Add icon to Start Menu**

Congratulations – you now have Cygwin installed with Vim!

Installing Cygwin packages

If you ever need to install additional packages in Cygwin, just rerun the installer while selecting the packages you want.

Using Cygwin

Open Cygwin – the program will be called **Cygwin64 Terminal** or **Cygwin Terminal**, depending on the version of your system, as can be seen in the following screenshot:

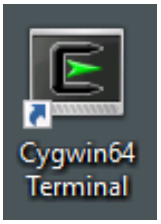

Figure 1.11 – The Cygwin64 Terminal application icon

Open it! You will see the following prompt, which will be familiar to Linux users:

Figure 1.12 – The Cygwin command prompt for the user called ruslan and the RUSLAN-DESKTOP machine

Cygwin supports all of the Unix-style commands we will be using in this book. This book will also say whether any commands need to be changed to work with Cygwin.

Type `vim` and hit *Enter* to start Vim, as demonstrated in the following screenshot:

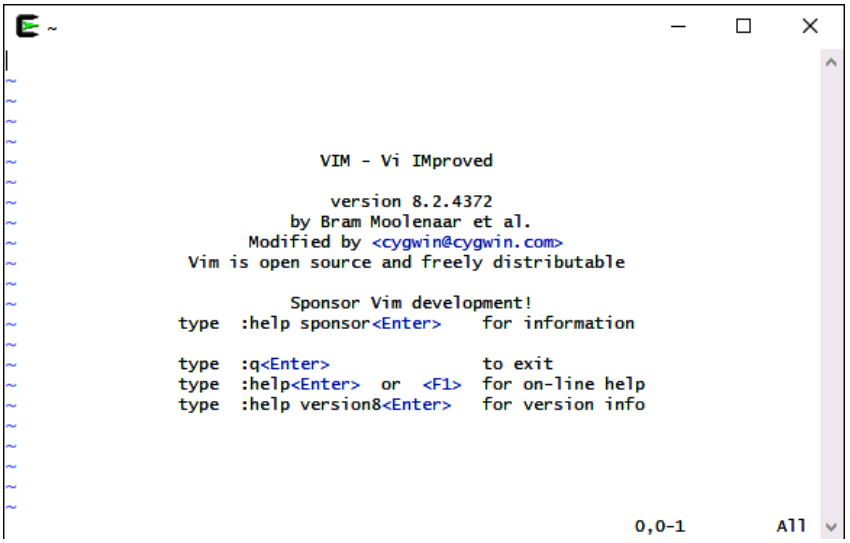

Figure 1.13 – Vim is installed through Cygwin (note Modified by <cygwin@cygwin.com>)

Cygwin is a way to get a Linux-like shell experience on Windows, meaning you'll have to follow Linux-specific instructions throughout this book if you decide to use Cygwin.

You'll also want to be careful with Windows-style line endings versus Linux-style line endings, as Windows and Linux treat line endings differently. If you run into an odd issue with Vim complaining about `^M` characters it is unable to recognize, run the `dos2unix` utility on the offending file to resolve the issue.

Visual Vim with gVim

You can read more about the graphical version of Vim in the *Vanilla Vim versus gVim* section later in this chapter.

As it always is with Windows, the process is slightly more visual. Navigate to `github.com/vim/vim-win32-installer` in your browser and download an executable installer. At the moment of writing this chapter, July 2023, the latest available version of Gvim is **9.0**.

> **You can use `winget` instead**
>
> Alternatively, if you're familiar with the `winget` tool, you can run `winget install -e --id vim.vim` (substitute with `vim.vim.nightly` if you'd like to live on the bleeding edge).

Open the executable and follow the prompts on the screen, as demonstrated by the following screenshot:

Figure 1.14 – gVim 9.0 setup welcome screen

Let's go ahead and hit **Next**, then **I Agree** until we arrive at the **Installation Options** screen. We're happy with most of the default options gVim has to offer, except that you might want to enable **Create .bat files for command line use**. This option will make the `vim` command work in the Windows Command Prompt. Some examples in this book rely on having a Command Prompt, so, enabling this option would help you follow along.

Here's a screenshot of the **Installation Options** screen with the proper boxes checked off:

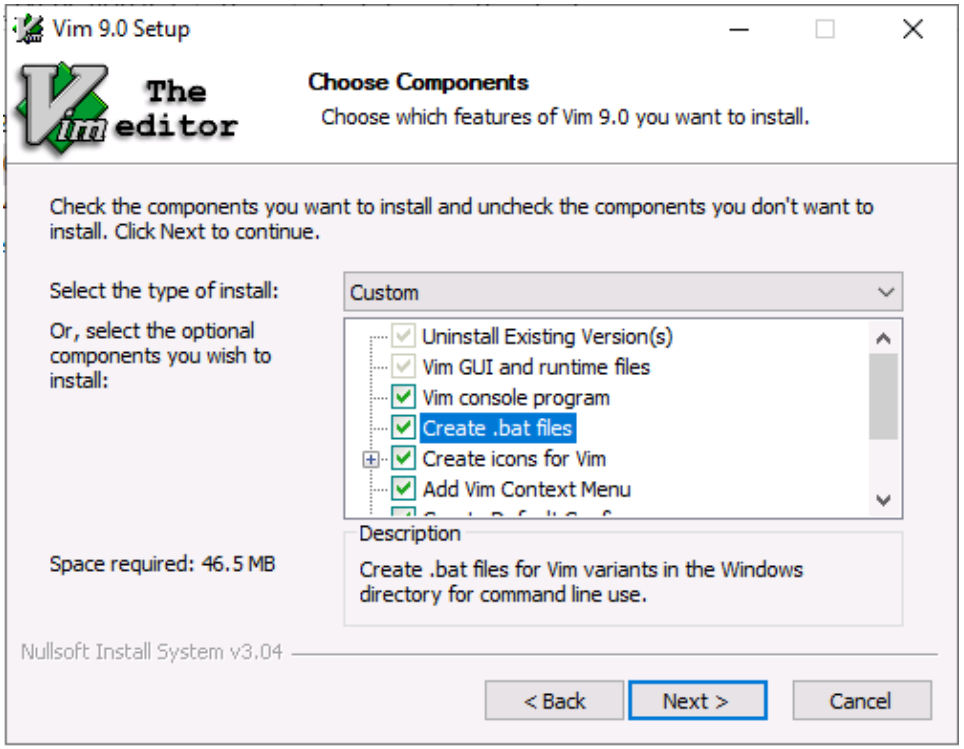

Figure 1.15 – gVim installation screen – note that Create .bat files is selected

Hit **Next >**. You'll want to continue with the following settings:

- **Select the type of install:** `Typical` (after **Create .bat files for command line use** is enabled, the type of install value changes to `Custom` automatically)
- **Do not remap keys for Windows behavior**
- **Right button has a popup menu, left button starts visual mode**
- **Destination Folder:** `C:\Program Files (x86)\Vim` (or a recommended default)

Once you're done, hit **Install** and then **Close**. Say **No** to the request to see the `README` file (or say **Yes** – I'm a book, not a cop).

You will now have a few icons pop up on your desktop, the most interesting one being **gVim 9.0** as shown in the following screenshot:

Figure 1.16 – The gVim 9.0 application icon

Start it, and you're ready to proceed! Happy Vimming!

Setting up on ChromeOS

ChromeOS has become increasingly popular – especially in the education sector and in a fraction of corporate settings. Chrome devices are cheap, user friendly, and can run on a potato with a couple of wires sticking out of it. I have been using a Chromebook as a daily driver for work for a couple of years now – and, thankfully, you don't have to abandon having Vim if ChromeOS is a platform of choice for you.

To run Vim on your ChromeOS device, you need to be able to install a Linux environment alongside it. It's not a complicated procedure, doesn't require dual booting, and provides seamless integration between ChromeOS and a Linux environment.

> **Linux support in ChromeOS**
>
> Some legacy ChromeOS devices don't support running a Linux environment, and you can check whether your device supports Linux by opening the **Settings** app and searching for `Linux`. If this yields no results, you won't be able to use Vim on your ChromeOS device.

First, set up Linux to run on ChromeOS by opening the **Settings** app, and navigating through the **Advanced** | **Developers** | **Linux development environment** | **Turn on** options:

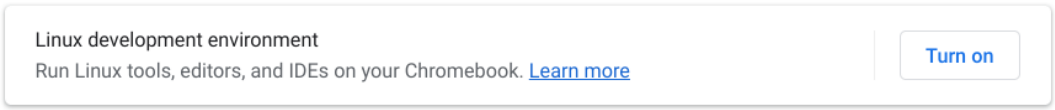

Figure 1.17 – A prompt to turn on a Linux development environment on a Chromebook

You'll be greeted by a prompt to set up a Linux development environment. Hit **Next** to continue. The default settings are sufficient, as demonstrated in the following screenshot:

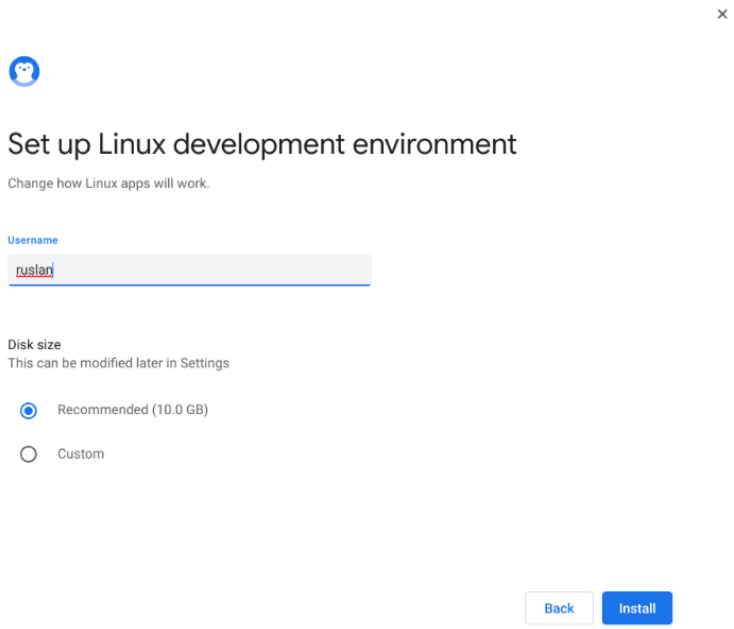

Figure 1.18 – The setup screen for a Linux development environment – it's fine to go with the recommended settings

Hit **Install**, and the whole setup should be over in under a couple of minutes.

The newly installed Debian Linux environment is accessible through the **Terminal** app. The default environment name is **penguin**, so you just need to select that:

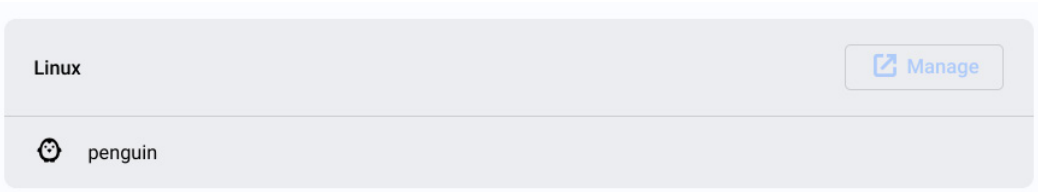

Figure 1.19 – Linux environment titled penguin installed on a ChromeOS

You'll see a (hopefully familiar) Linux command-line prompt:

Figure 1.20 – Linux terminal within ChromeOS (a silly cowsay command can be installed by running $ sudo apt install cowsay)

Vim comes preinstalled on a vast majority of Linux machines, and this one is not an exception: you should already have access to Vim.

> **Compile Vim for the latest version**
>
> If you'd like to have the latest version of Vim, I recommend compiling Vim from its official Git repository (it's easy). Visit the *Setting up on Linux* section earlier in this chapter for instructions on installing the latest Vim version.

Now, you can launch Vim by invoking it from the terminal:

```
$ vim
```

This is the output as shown here:

Figure 1.21 – Vim welcome screen, as seen in the ChromeOS terminal

Access ChromeOS files through Linux

Due to integration between ChromeOS and Linux, you can access your Linux home directory (accessible via the `cd ~` command from the Terminal window) via the ChromeOS Files app:

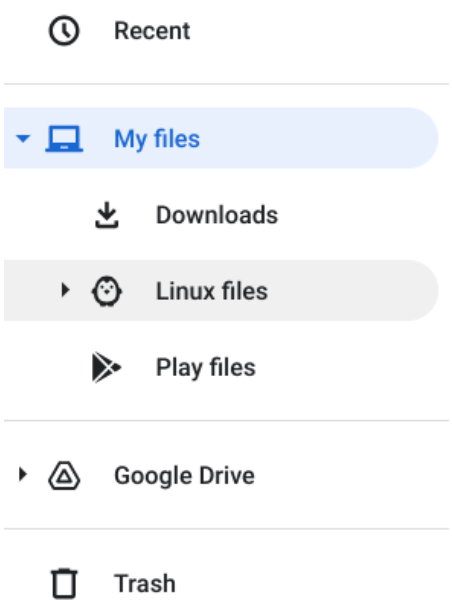

Figure 1.22 – You can access Linux files through the ChromeOS **Files** app

In addition, you can share individual ChromeOS files with Linux by right-clicking on the file and selecting **Share with Linux**. This makes the files accessible in Linux via the `/mnt/chromeos` directory.

Verifying and troubleshooting the installation

Regardless of the platform you use to install Vim, it's good to make sure that, with Vim, all the right features are enabled. On a command line, run the following command:

```
$ vim --version
```

You will see the following output, with a set of features having a + and a - in front of them:

```
~$ vim --version
VIM - Vi IMproved 9.0 (2022 Jun 28, compiled Sep 01 2022 21:10:19)
Included patches: 1-354
Compiled by Arch Linux
Huge version without GUI.  Features included (+) or not (-):
+acl               +file_in_path      +mouse_urxvt       -tag_any_white
+arabic            +find_in_path      +mouse_xterm       +tcl/dyn
+autocmd           +float             +multi_byte        +termguicolors
+autochdir         +folding           +multi_lang        +terminal
-autoservername    -footer            -mzscheme          +terminfo
-balloon_eval      +fork()            +netbeans_intg     +termresponse
+balloon_eval_term +gettext           +num64             +textobjects
-browse            -hangul_input      +packages          +textprop
++builtin_terms    +iconv             +path_extra        +timers
+byte_offset       +insert_expand     +perl/dyn          +title
+channel           +ipv6              +persistent_undo   -toolbar
+cindent           +job               +popupwin          +user_commands
-clientserver      +jumplist          +postscript        +vartabs
-clipboard         +keymap            +printer           +vertsplit
+cmdline_compl     +lambda            +profile           +vim9script
+cmdline_hist      +langmap           +python            +viminfo
+cmdline_info      +libcall           -python3           +virtualedit
```

Figure 1.23 – Output of the vim --version command

In the preceding screenshot, you can see that my Vim was actually compiled with Python 2 support (`+python`) instead of Python 3 support (`-python3`). To correct the issue, I'd have to either recompile Vim with `+python3` enabled (for which I'd have to install required dependencies) or find a package that distributes a compiled version of Vim with `+python3` enabled.

Vim integrations

Note that `+python3` and similar options refer to Python 3 integration of Vim. You can still edit any file you want (including Python 3), but it does mean your Vim instance, plugins, and scripts won't be able to run Python 3.

Ask for `:help`

For a list of all features Vim can have enabled, see `:help feature-list`.

For instance, if we wanted to recompile Vim with Python 3 support on Linux, we would do the following:

```
$ git clone https://github.com/vim/vim.git
$ cd vim/src
$ ./configure --prefix=/usr/local \
    --with-features=huge \
    --enable-python3interp
$ make
$ sudo make install
```

Specifying feature sets

We're passing the `--with-features=huge` flag in order to compile Vim with most features enabled. However, `--with-features=huge` does not install language bindings, so we need to explicitly enable Python 3.

In general, if your Vim is not behaving like other Vim installations (including the behavior described in this book), you might be missing a feature.

Depending on your system and the features you require, the process might be slightly or vastly different. A quick web search along the lines of `Installing Vim <version> with +<feature> on <operating system>` should help.

Now that you're through the installation instructions, let's look a little closer at what we've installed.

Vanilla Vim versus gVim

Using the instructions given before, you've installed two flavors of Vim: command-line Vim, and gVim. This is how gVim looks in Windows:

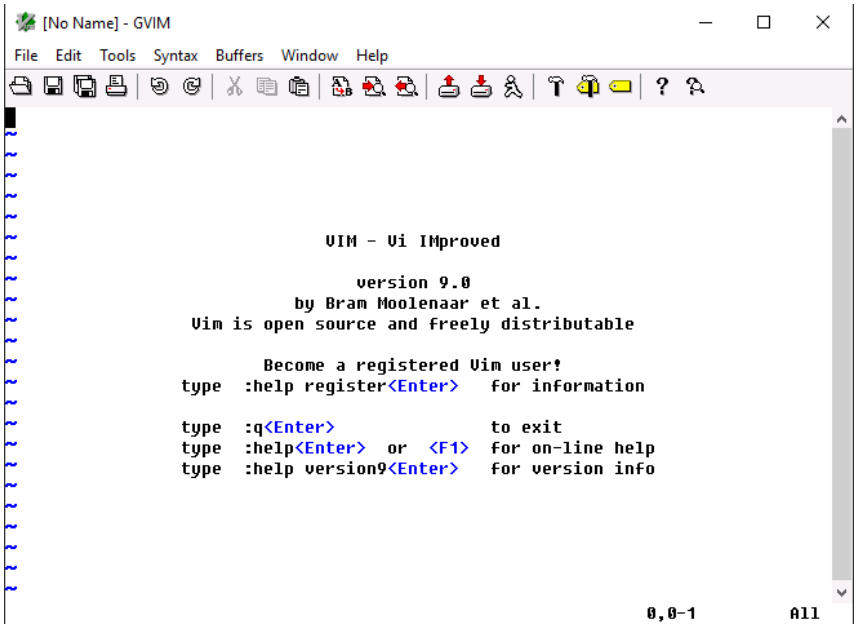

Figure 1.24 – gVim interface in Windows

The gVim hooks up a **graphical user interface** (**GUI**) to Vim, has better mouse support, and adds more context menus. It also supports a wider range of colors than many terminal emulators and provides some quality-of-life features you'd expect from a modern GUI.

You can launch gVim by running the `gVim 9.0` executable on Windows, or on Linux and macOS by invoking the following command:

```
$ gvim
```

Windows users might favor gVim.

This book focuses on increasing the effectiveness of one's text editing skills, so we will shy away from navigating multiple menus in gVim, as these are rather intuitive, and take the user out of the flow.

Hence, we will focus on a non-graphical version of Vim, but everything that's applicable to Vim also applies to gVim. The two share configurations, and you can swap between the two as you go. Overall, gVim is slightly more newbie friendly, but it doesn't matter which one you choose to use for the purpose of this book.

Try both!

Configuring Vim with your .vimrc

Vim reads configuration from a `.vimrc` file. Vim works out of the box, but there are certain options that make working with code a lot easier.

> **Spot hidden files**
>
> In Unix-like systems, files that start with a period (`.`) are hidden. To see them, run `ls -a` in a Command line.

In Linux and macOS, `.vimrc` is located in your `user` directory (the full path would be `/home/<username>/.vimrc`). You can also find your `user` directory by opening a Command Prompt and running the following command:

```
$ echo $HOME
```

Older versions of Windows Explorer did not allow periods in file names, so the file is named `_vimrc`. It's usually located in `C:\Users\<username>\_vimrc`, but you can also locate it by opening the Windows Command Prompt and running the following command:

```
$ echo %USERPROFILE%
```

> **Locating `.vimrc`**
>
> If you run into problems, open Vim and type in `:echo $MYVIMRC` followed by *Enter*. It should display where Vim is reading `.vimrc` from.

Find the proper location for your OS, and place the prepared configuration file there. You can download the `.vimrc` file used for this chapter from GitHub at `https://github.com/PacktPublishing/Mastering-Vim-Second-Edition/tree/main/Chapter01`. The following code shows the contents of a `.vimrc` file used in this chapter:

```
syntax on " Enable syntax highlighting.
filetype plugin indent on " Enable file type based options.

set nocompatible " Don't run in backwards compatible mode.

set autoindent " Respect indentation when starting new line.
set expandtab " Expand tabs to spaces. Essential in Python.
set tabstop=4 " Number of spaces tab is counted for.
set shiftwidth=4 " Number of spaces to use for autoindent.

set backspace=2 " Fix backspace behavior on most terminals.

colorscheme murphy " Change a colorscheme.
```

You'll eventually want to change these

For extra credit, you may want to limit some options to Python only – especially if you're (hopefully) planning to use Vim as your primary editor for multiple file types. I recommend you prefix certain options with `autocmd filetype python`, which will only apply options to Python files:

```
autocmd filetype python set expandtab
autocmd filetype python set tabstop=4
autocmd filetype python set shiftwidth=4
```

Lines starting with a double quote (`"`) are comments and are ignored by Vim. These settings bring in some sensible defaults, such as syntax highlighting and consistent indentation. They also fix one of the common sticking points in a bare-bones Vim installation – inconsistent backspace key behavior across different environments.

Trying out the configuration options

When working with Vim configuration, you can try things out before adding them to your `.vimrc` file. To do that, type `:` followed by a command, for example, `:set autoindent` (press *Enter* to execute). If you ever want to know the value of a setting, add `?` at the end of the command: for example, `:set tabstop?` will tell you the current `tabstop` value.

I've also changed `colorscheme` to make screenshots look better in print, but you don't have to (unless you're writing a book about Vim – then you probably should).

> **The world is full of colors**
>
> Vim 9 comes prepackaged with many color themes, and the list grows with every release. You can try out a color theme by typing `:colorscheme <name>` and hitting *Enter*, and you can cycle through the available color scheme names by typing `:colorscheme` followed by a space and by hitting *Tab* multiple times. You can read more about configuring Vim and color schemes (including using color schemes you find online) in *Chapter 7*.

You've configured Vim, and now it's time to learn how to actually use it!

Common operations (or, how to exit Vim)

We will now focus on interacting with Vim without the use of a mouse or navigational menus. Here's a meme I found some years back:

Figure 1.25 – An accurate portrayal of a typical Vim user (source: https://twitter.com/iamdevloper/status/435555976687923200)

Programming is a focus-intensive task on its own. Hunting through context menus is nobody's idea of a good time, and keeping our hands on the home row of your keyboard helps trim constant switching between a keyboard and a mouse.

In this section, we'll learn how to open (and, more importantly, close) Vim, save files, and make basic edits.

Opening files

First, start your favorite Command Prompt (Terminal in Linux and macOS, Cygwin in Windows). We'll be working on a very basic Python application. For simplicity's sake, let's make a simple square root calculator. Run the following command:

```
$ vim spam.py
```

GUI for the win!

If you're using gVim, you can open a file by going into a **File** menu and choosing **Open**. Sometimes, a graphical interface is exactly what you need!

This opens a file named `spam.py`. If the file existed, you'd see its contents here, but since it doesn't, we're greeted by an empty screen, as shown in the following example:

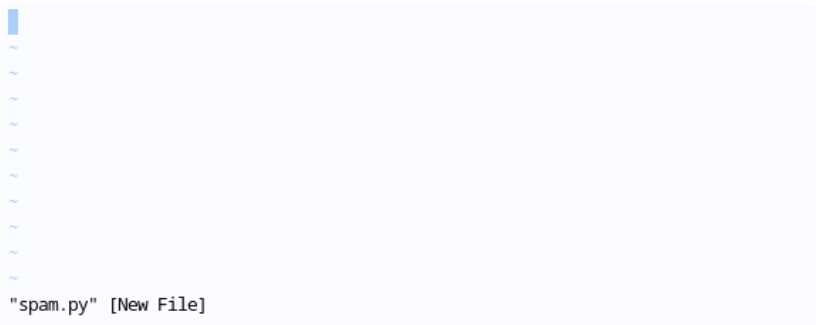

Figure 1.26 – Opening a new file in Vim as indicated by the `[New File]` text

You can tell that the file doesn't exist by the `[New File]` text next to a file name at the bottom of the screen. Woohoo! You've just opened your first file with Vim!

If you already have Vim open, you can load a file by typing the following, and hitting *Enter*:

```
:e spam.py
```

You have just executed your first Vim command! Pressing the colon character (`:`) enters a command-line mode, which lets you enter a line of text that Vim will interpret as a command. Commands are terminated by hitting the *Enter* key, which allows you to perform various complex operations, as well as accessing your system's Command line. The `:e` command stands for *edit*.

Vim help often refers to the *Enter* key as a `<CR>`, which stands for carriage return.

Changing text

By default, you're in Vim's normal mode, meaning that every key press corresponds to a particular command. Hit *i* on your keyboard to enter an insert mode. This will display **-- INSERT --** in a status

line (at the bottom), and, if you're using gVim, it will change the cursor from a block to a vertical line, as can be seen in the following example:

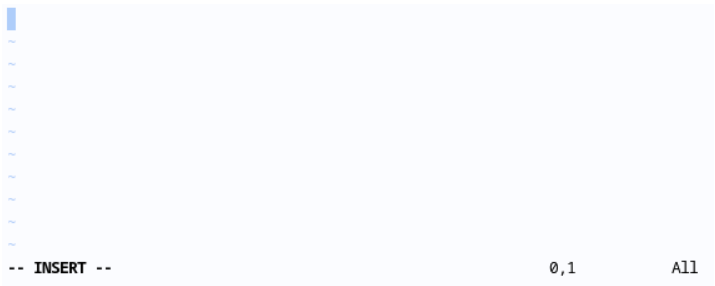

Figure 1.27 – Vim in insert mode, as indicated by -- INSERT -- in the bottom-left corner

The insert mode behaves just like any other modeless editor. Normally, we wouldn't spend a lot of time in insert mode except for adding new text.

> **Modes, modes, modes**
>
> You've already encountered three of Vim's modes: command-line mode, normal mode, and insert mode. This book will cover more modes – see *Chapter 3* for details and explanation.

Let's create our Python application by typing in the following code. We'll be navigating this little snippet throughout this chapter:

```
#!/usr/bin/python

import random

INGREDIENTS = ['egg', 'sausage', 'bacon', 'ham', 'crumpets', 'spam']

def prepare_menu_item(ingredient, with_spam=True):
    if with_spam:
        return 'spam ' + ingredient
    return ingredient

def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for ingredient in INGREDIENTS:
        has_spam = random.choice([True,  False])
        menu.append(prepare_menu_item(ingredient, with_spam=has_spam))
    print('Waitress: Well, there\'s', ', '.join(menu))


if __name__ == '__main__':
    main()
-- INSERT --
```

Figure 1.28 – A simple Python 3 program referencing Monty Python's "Spam" sketch

To get back to normal mode in Vim, hit *Esc* on your keyboard. You'll see that -- **INSERT** -- has disappeared from the status line. Now, Vim is ready to take commands from you again!

> **This code isn't very good**
>
> The preceding code does not display Python best practices and is provided to illustrate some of Vim's capabilities.

Saving and closing files

Let's save our file! Execute the following command:

```
:w
```

> **Don't forget to execute the command!**
>
> Don't forget to hit *Enter* at the end of a command to execute it.

`:w` stands for **write**.

> **Naming files**
>
> The write command can also be followed by a filename, making it possible to write to a different file, other than the one that is open (`:w spam_2.py`). To change the currently open file to a new one when saving, use the `:saveas` command: `:saveas spam_2.py`.

Let's exit Vim and check whether the file was indeed created. `:q` stands for **quit**. You can also combine write and quit commands to write and exit by executing `:wq`.

```
:q
```

If you made changes to a file and want to exit Vim without saving the changes, you'll have to use `:q!` to force Vim to quit. The exclamation mark at the end of the command forces its execution.

> **Shortening commands**
>
> Many commands in Vim have shorter and longer versions. For instance, `:e`, `:w`, and `:q` are short versions of `:edit`, `:write`, and `:quit`. In the Vim manual, the optional part of the command is often annotated in square brackets (`[]`); for example, `:w[rite]` or `:e[dit]`.

Now that we're back in our system's command line, let's check the contents of a current directory, as seen in the following code:

```
$ ls
$ python3 spam.py
```

The following screenshot shows what the two preceding commands should output:

```
~/Mastering-Vim-Second-Edition/Chapter01$ ls -a
.  ..  README.md  spam.py  .vimrc
~/Mastering-Vim-Second-Edition/Chapter01$ python3 spam.py
Scene: A cafe. A man and his wife enter.
Man: Well, what've you got?
Waitress: Well, there's spam egg, sausage, bacon, ham, crumpets, spam spam
~/Mastering-Vim-Second-Edition/Chapter01$
```

Figure 1.29 – Output of the ls -a and python3 spam.py commands

Command spotlight

In Unix, `ls` lists the contents of a current directory (the `-a` flag shows hidden files). `python3 spam.py` executes the script using a Python 3 interpreter.

A word about swap files

By default, Vim keeps track of the changes you make to files in swap files. The swap files are created as you edit the files, and are used to recover the contents of your files in case either Vim, your SSH session, or your machine crashes. If you don't exit Vim cleanly, or try to edit the same file multiple times at the same time, you'll be greeted by the following screen:

```
E325: ATTENTION
Found a swap file by the name ".spam.py.swp"
          owned by: ruslano   dated: Thu Jul 27 17:12:33 2023
         file name: ~ruslano/Mastering-Vim-Second-Edition/Chapter01/spam.py
          modified: YES
         user name: ruslano   host name: penguin
        process ID: 1719 (STILL RUNNING)
While opening file "spam.py"
             dated: Thu Jul 27 17:02:10 2023

(1) Another program may be editing the same file.  If this is the case,
    be careful not to end up with two different instances of the same
    file when making changes.  Quit, or continue with caution.
(2) An edit session for this file crashed.
    If this is the case, use ":recover" or "vim -r spam.py"
    to recover the changes (see ":help recovery").
    If you did this already, delete the swap file ".spam.py.swp"
    to avoid this message.

Swap file ".spam.py.swp" already exists!
[O]pen Read-Only, (E)dit anyway, (R)ecover, (Q)uit, (A)bort:
```

Figure 1.30 – A swap file error when attempting to open a file

You can either hit *r* to recover the swap file contents or *d* to delete the swap file and dismiss the changes. If you decide to recover the swap file, you can prevent the same message from showing up the next time you open the file in Vim by reopening a file, running `:e`, and pressing *d* to delete the swap file (although Vim won't let you delete the swap file if the file in question is currently open in another Vim instance).

By default, Vim creates files such as `<filename>.swp` and `.<filename>.swp` in the same directory as the original file. If you don't like your file system being littered with swap files, you can change this behavior by telling Vim to place all the swap files in a single directory. To do so, add the following to your `.vimrc`:

```
set directory=$HOME/.vim/swap//
```

Note that double directory delimiter at the end; this setting won't work correctly without it!

> **For Windows users**
>
> If you're on Windows, you should use `set directory=%USERDATA%\.vim\swap//` (note the direction of the last two slashes).

You can also choose to disable the swap files completely by adding `set noswapfile` to your `.vimrc`.

"But," I hear you say, "I spend most of my time navigating code (or text) rather than writing it top to bottom!" Never fret, as this is what sets Vim apart from conventional editors. The next section will teach you to navigate around what you just wrote.

Moving around – talk to your editor

Vim allows you to navigate content a lot more efficiently than most conventional editors. Let's start with the basics.

You can move your cursor around, character by character, by using arrow keys or the *h*, *j*, *k*, and *l* keys. This is the least efficient and the most precise way to move:

Key	Alternative key	Action
h	Left arrow	Move cursor left
j	Down arrow	Move cursor down
k	Up arrow	Move cursor up
l	Right arrow	Move cursor right

The following diagram is a visual representation that might be a little easier on the eyes:

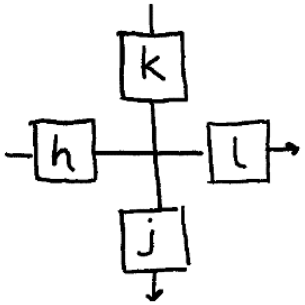

Figure 1.31 – Visual representation of the h, j, k, and l directional keys

Vi (Vim's predecessor) was created on an old ADM-3A terminal, which didn't have arrow keys. The *h*, *j*, *k*, and *l* keys were used as arrows.

Figure 1.32 – The ADM-3A terminal (image by Chris Jacobs, Wikipedia (CC BY-SA 3.0))

Try it! There's a lot of value to getting used to the *h*, *j*, *k*, and *l* keys for movement; for example, your hands stay on the home row of your keyboard. This way you don't have to move your hands and it helps you stay in the flow. Furthermore, many applications treat *h*, *j*, *k*, and *l* as arrow keys – you'd be surprised how many tools respond to these.

Now, you might be inclined to hit directional keys multiple times to get to a desired position, but there's a better way! You can prefix every command with a number, which would repeat the command that number of times. For example, hitting *5* + *j* will move the cursor five lines down, while hitting *1* + *4* + *l* will move the cursor 14 characters to the right. This works with most commands you encounter in this book.

Calculating the exact number of characters you would like to move is pretty hard (and nobody wants to do it), so there's a way to move by words. Use *w* to move to the beginning of the next **w**ord, and use *e* to get to the **e**nd of the closest word. To move **b**ackward to the beginning of the word, hit *b*.

You can also capitalize these letters to treat everything but a white space as a word! This allows you to differentiate between the kind of things you'd like to traverse.

Vim has two kinds of word objects: referred to as lowercase "**word**" and uppercase "**WORD**". In the Vim world, **word** is a sequence of letters, digits, and underscores. **WORD** is a sequence of any non-blank characters separated by white space (it's technically more complicated than that; see `:help iskeyword` if you're curious).

Let's take the following line of code from our example:

```
prepare_menu_item(ingredient, with_spam=has_spam)
```

> **Note**
> Notice the cursor position – it's hovering over the first character of `ingredient`.

Hitting *w* will move the cursor to the next comma while hitting *W* will take you to the beginning of `with_spam`. Capitalized *W*, *E*, and *B* will treat any characters bundled together and separated by a space as their own words. This can be seen in the following table:

Key	Action
w	Move forward by **word**
e	Move forward until the end of the **word**
W	Move forward by **WORD**
E	Move forward until the end of the **WORD**
b	Move backward to the beginning of the **word**
B	Move backward to the beginning of the **WORD**

The following screenshot shows more examples of how each command behaves:

Key	Initial cursor position	Resulting cursor position
w	`prepare_menu_item(ingredient, with_spam=has_spam)`	`prepare_menu_item(ingredient, with_spam=has_spam)`
e	`prepare_menu_item(ingredient, with_spam=has_spam)`	`prepare_menu_item(ingredient, with_spam=has_spam)`
b	`prepare_menu_item(ingredient, with_spam=has_spam)`	`prepare_menu_item(ingredient, with_spam=has_spam)`
W	`prepare_menu_item(ingredient, with_spam=has_spam)`	`prepare_menu_item(ingredient, with_spam=has_spam)`
E	`prepare_menu_item(ingredient, with_spam=has_spam)`	`prepare_menu_item(ingredient, with_spam=has_spam)`
B	`prepare_menu_item(ingredient, with_spam=has_spam)`	`prepare_menu_item(ingredient, with_spam=has_spam)`

Combine the movements shown with the directional movements you learned earlier to move in fewer keystrokes!

It's also really useful to move in paragraphs. Everything separated by at least two new lines is considered a paragraph, which also means each code block is a paragraph, as can be seen in the following example:

```
INGREDIENTS = ['egg', 'sausage', 'bacon', 'ham', 'crumpets', 'spam']

def prepare_menu_item(ingredient, with_spam=True):
    if with_spam:
        return 'spam ' + ingredient
    return ingredient

def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for ingredient in INGREDIENTS:
        has_spam = random.choice([True,  False])
        menu.append(
                prepare_menu_item(ingredient, with_spam=has_spam)
        )
    print('Waitress: Well, there\'s', ', '.join(menu))
```

Figure 1.33 – You can see three paragraphs here

The `INGREDIENTS` constant, and the `prepare_menu_item` and `main` functions are three separate paragraphs. Use a closing curly brace, *}*, to move forward, and an opening curly brace, *{*, to move backward, as detailed in the following table:

Command	Action
{	Move back by one paragraph
}	Move forward by one paragraph

Don't forget to combine these two with numbers if you need to move by more than one paragraph.

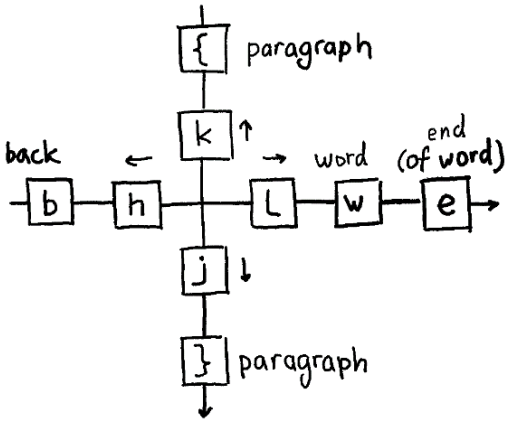

Figure 1.34 – A visual representation of basic movement keys in Vim

There are more ways to move around, but these are the most important basics. We'll be covering more complex ways to navigate in *Chapter 2*.

Making simple edits in insert mode

When working with Vim, you usually want to spend as little time as possible in the insert mode (unless you're writing and not editing). Since most text operations involve editing, we'll focus on that.

You've already learned to enter the insert mode by pressing *i*. There are more ways to get to the insert mode. Often, you will want to change some piece of text for another one, and there's a command just for that – *c*. The change command allows you to remove a portion of text and immediately enter an insert mode. Change is a compound command, meaning that it needs to be followed by a command that tells Vim what needs to be changed. You can combine it with any of the movement commands you've learned before. Here are some examples:

Command	Before	After
Cw	`prepare_menu_item(ingredient, with_spam=has_spam)`	`(ingredient, with_spam=has_spam)`
c3e	`prepare_menu_item(ingredient, with_spam=has_spam)`	`, with_spam=has_spam)`
Cb	`prepare_menu_item(ingredient, with_spam=has_spam)`	`prepare_menu_item(ent, with_spam=has_spam)`
c4l	`prepare_menu_item(ingredient, with_spam=has_spam)`	`are_menu_item(ingredient, with_spam=has_spam)`
cW	`prepare_menu_item(ingredient, with_spam=has_spam)`	`with_spam=has_spam)`

words versus WORDS

Remember, non-alphanumeric characters (including punctuation) are treated as **words**. For example, `prepare_menu_item(ingredient, has_spam)` will be considered six **words** (but only one **WORD**): `prepare_menu_item`, `(`, `ingredient`, `,` (comma), `has_spam`, and `)`.

Exception

As an odd exception, *cw* behaves like *ce*. This is a leftover from Vi, Vim's predecessor.

As you learn more complex movement commands, you can combine these with a change for quick and seamless editing. We'll also be covering a few plugins that will supercharge a change command to allow for even more powerful editing, such as changing text within braces or replacing the type of quotes on the go.

Just like a sentence

All of these examples follow the `<command> <number> <movement or a text object>` structure. You can put a number before or after `<command>`.

For example, say you wish to change the following line:

```
prepare_menu_item(ingredient, with_spam=has_spam)
```

Say you want to change the `ingredient` to `'egg'`:

```
prepare_menu_item('egg', with_spam=has_spam)
```

To accomplish that, you can execute the following set of commands:

Contents of the line	Action
`prepare_menu_item(ingredient, with_spam=has_spam)`	Start with a cursor at the beginning of the line
`prepare_menu_item(ingredient, with_spam=has_spam)`	Hit `2w` to move the cursor three words forward to the beginning of `ingredient`
`prepare_menu_item(, with_spam=has_spam)`	Press `cw` to delete the `ingredient` word and enter the insert mode
`prepare_menu_item('egg', with_spam=has_spam)`	Type `'egg'`
`prepare_menu_item('egg', with_spam=has_spam)`	Hit the Esc key to return to NORMAL mode

Sometimes, we just want to cut things, without putting anything instead, and *d* does just that. It stands for delete. It behaves similarly to *c*, except that the behavior of *w* and *e* is more standard, as can be seen in the following example:

Command	Before	After
dw	`prepare_menu_item(ingredient, with_spam=has_spam)`	`prepare_menu_item(, with_spam=has_spam)`
d3e	`prepare_menu_item(ingredient, with_spam=has_spam)`	`, with_spam=has_spam)`
db	`prepare_menu_item(ingredient, with_spam=has_spam)`	`prepare_menu_item(ent, with_spam=has_spam)`
d5l	`prepare_menu_item(ingredient, with_spam=has_spam)`	`prepare_menu_item(ingredient, spam=has_spam)`
dW	`prepare_menu_item(ingredient, with_spam=has_spam)`	`prepare_menu_item(ingredient,`

There are also two more nifty shortcuts that allow you to change or delete a whole line:

Command	What it does
cc	Clears the whole line and enters insert mode. Preserves current indentation level, which is useful when coding.
dd	Deletes an entire line.

For example, look at the following piece:

```
def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for ingredient in INGREDIENTS:
        has_spam = random.choice([True,  False])
        menu.append(prepare_menu_item(ingredient, with_spam=has_spam))
    print('Waitress: Well, there\'s', ', '.join(menu))
```

Figure 1.35 – A piece of code we wrote (or copied) earlier.

By hitting *dd* you will completely remove a line, as demonstrated in the following example:

```
def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for ingredient in INGREDIENTS:
        has_spam = random.choice([True,  False])
    print('Waitress: Well, there\'s', ', '.join(menu))
```

Figure 1.36 – *dd* removes a line and places the cursor on the next line

Hitting *cc* will clear the line and enter insert mode with the proper indent, as shown in the following example:

```
def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for ingredient in INGREDIENTS:
        has_spam = random.choice([True,  False])

    print('Waitress: Well, there\'s', ', '.join(menu))
```

Figure 1.37 – *cc* clears the line, and puts you into insert mode at the beginning of the line.

Visual mode

If you run into difficulties picking the right movement commands, you can also use the visual mode to select the text you want to modify. Hit *v* to enter the visual mode and use the usual movement commands to adjust the selection. Run the desired command (such as *c* to change or *d* to delete) once you're satisfied with the selection.

Persistent undo and repeat

Like any editor, Vim keeps track of every operation. Press *u* to undo a last operation, and *Ctrl* + *r* to redo it.

Undo tree

To learn more about Vim's undo tree (Vim's undo history is not linear) and how to navigate it, see *Chapter 4*.

Vim also allows you to persist undo history between sessions, which is great if you want to undo (or remember) something you've done a few days ago!

You can enable persistent undo by adding the following line to your `.vimrc`:

```
set undofile
```

However, this will litter your system with an undo file for each file you're editing. You can consolidate the undo files in a single directory, as seen in the following example:

```
" Set up persistent undo across all files.
set undofile
let my_undo_dir = expand('$HOME/.vim/undodir')
```

```
if !isdirectory(my_undo_dir))
  call mkdir(my_undo_dir, "p")
endif
set undodir=my_undo_dir
```

> **For Windows users**
> If you're using Windows, replace the directories with `$USERPROFILE\vimfiles\undodir` (and you'll be making changes to `_vimrc` instead of `.vimrc`).

Now, you'll be able to undo and redo your changes across sessions.

Read the Vim manual using :help

The best learning tool Vim can offer is certainly a `:help` command, as can be seen in the following screenshot:

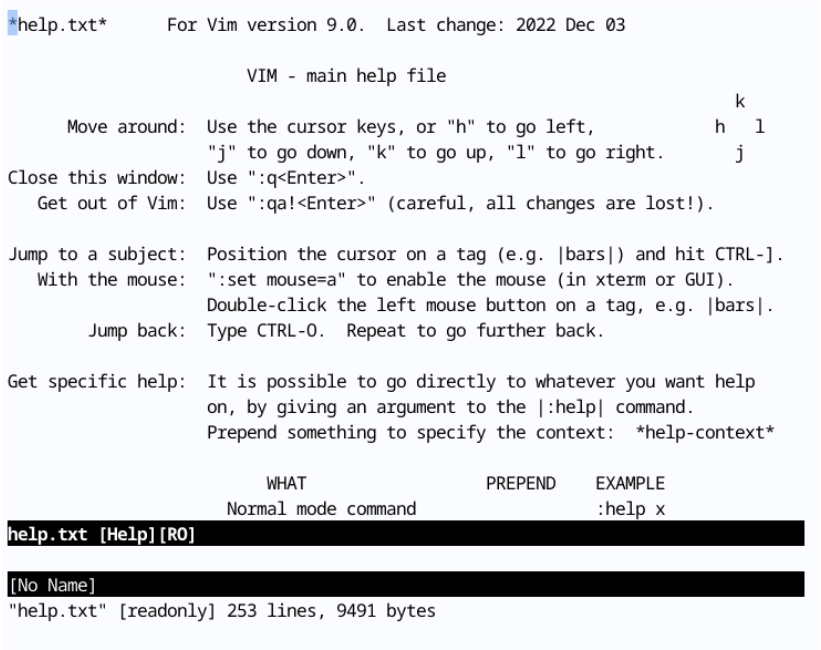

Figure 1.38 – The :help page – if you've read it, you probably wouldn't have needed to read this book

It's an enormous collection of resources and tutorials that comes installed with Vim. Scroll through using the *Page Up* and *Page Down* keys (bonus point for using *Ctrl* + *b* and *Ctrl* + *f* instead), there is a lot of useful information there.

`:help help-summary`

Consider reading through the `help-summary` page at some point in your Vim journey. It's short, and arms you with many tools needed to navigate Vim help effectively. I incorporated some tips from `help-summary` in this section, but there are more!

Whenever you are stuck or want to learn more about a particular command, try searching it using `:help` (you can shorten it to `:h`). Let's try searching for the normal mode `cc` command we've learned:

```
:h cc
```

Here's the output of this code:

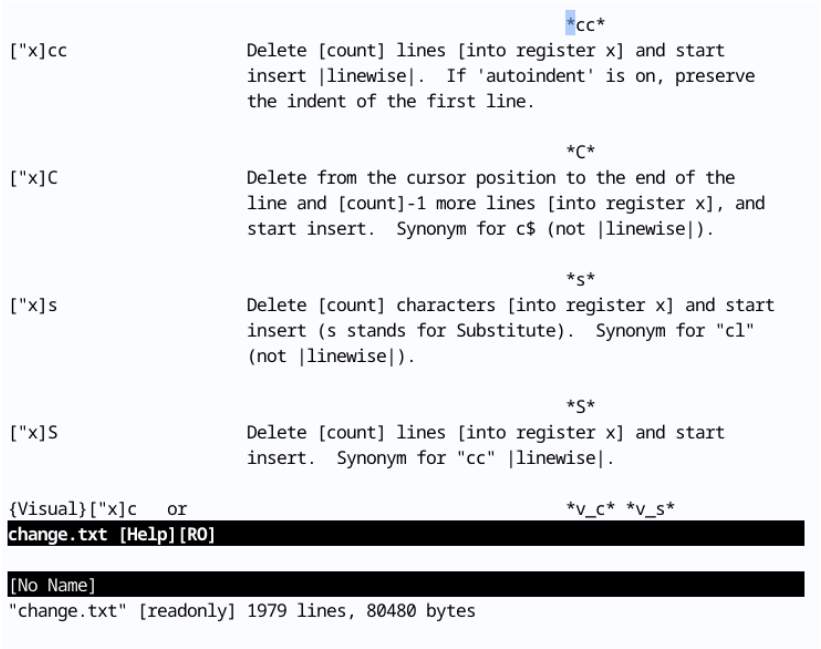

Figure 1.39 – A :help page for a cc command

Help tells us the way the command works, as well as how different options and settings affect the command (for instance, the `autoindent` setting preserves the indentation).

Searching for options

In Vim help, options are always denoted with single quotation marks. For example, if you wanted to look for the `autoindent` option (remember `set autoindent` in your `.vimrc`?), you'll execute `:help 'autoindent'`.

`:help` is a command that navigates a set of help files. As you look through the help files, you'll notice that certain words are highlighted in color. These are tags, and can be searched for using the `:help` command. Unfortunately, not every tag name is intuitive. For instance, if we wanted to learn how to search for a string in Vim, we could try using the following:

```
:h search
```

However, it looks like this command takes us to the entry on expression evaluation, which is not exactly what we were looking for, as demonstrated by the following screenshot:

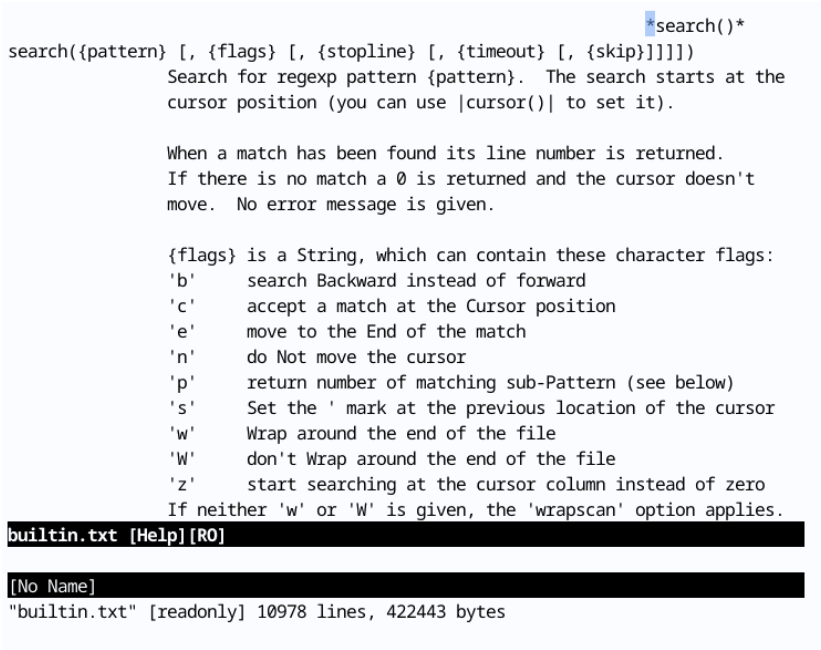

Figure 1.40 – A help page for a search() expression

To find the right entry, type `:h search` (don't hit *Enter* yet) followed by *Ctrl* + *d*. This will give you a list of help tags containing the `search` substring. One of the options shown is `search-commands`, which is what we'd be looking for. Complete your command in the following way to get to the entry we were looking for:

```
:h search-commands
```

The following display shows the right help entry for search:

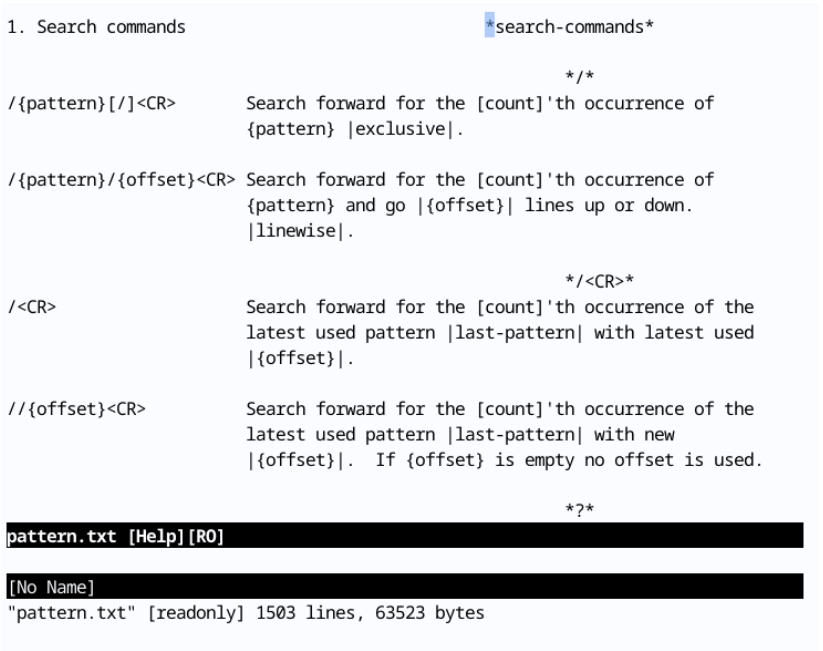

Figure 1.41 – A :help page for search commands

Navigating search files

A discerning reader might notice certain keywords adorned with various characters (`[`, `{`, `|`, etc.) or highlighted (depending on your color scheme) in help pages. When your cursor is over a keyword in a help file, you can press *Ctrl +]* to jump to the definition of the keyword, and *Ctrl + o* to jump back. Give it a shot!

Search

Speaking of search functionality, you can search inside help pages (or any file open in Vim) using `/search term` to search forward from the cursor or `?search term` to search backward. See *Chapter 2* to learn more about how to perform search operations.

Finally, Vim comes with a handy `vimtutor` utility, which you can invoke on the command line. This book covers most things that `vimtutor` teaches, but it doesn't mean you should sleep on it – `vimtutor` includes many exercises that can build Vim commands into your muscle memory.

Don't forget to use Vim's help system any time you have questions or want to better understand the way Vim behaves.

Summary

The original Vi was developed to work through remote terminals when bandwidth and speed were limited. These limitations guided Vi toward establishing an efficient and deliberate editing process, which is what's at the core of Vim—Vi Improved today.

Throughout this chapter, you've picked up a few (hopefully) interesting tidbits about the history of Vim and the difference between its major versions.

You've learned how to install and update Vim and its graphical counterpart – gVim – on every major platform (in more ways than you will ever need).

You've learned to configure your Vim through tinkering with `.vimrc`, which is something you will often go back to as you customize the editor for your own needs.

You've picked up the basics of working with files, moving around Vim, and making changes. Vim's concept of text objects (letters, words, paragraphs) and composite commands (such as *d2w* - **d**elete **2 w**ords) empower precise text operations.

If there's one thing you could take away from this chapter, it would be `:help`. Vim's internal help system is incredibly detailed, and it can answer most, if not all, questions you might have, as long as you know what you're looking for.

In the next chapter, we'll be looking into getting more out of Vim. You'll learn how to navigate files and get better at editing text.

2
Advanced Editing and Navigation

Throughout this chapter, you will get a lot more comfortable using Vim in your day-to-day tasks. You will be working with a Python code base, which should provide you with a set of real-life scenarios for working with code. If you have a project of your own handy, you can choose to try out the lessons taught in this chapter using your own project files; however, you might find that not every scenario applies to your code base.

The following topics will be covered in this chapter:

- A quick-and-dirty way of installing Vim plugins (with a better way to install and manage plugins following in *Chapter 3*, *Follow the Leader – Plugin Management*).
- Keeping your workspace organized when working with multiple or long files using buffers, windows, tabs, and folds
- Navigating complex file trees without leaving Vim with Netrw, NERDTree, Vinegar, or CtrlP
- Advanced navigation throughout a file, and covering more types of text objects: using `grep` and `ack` to look for things across files, and EasyMotion, a lightning-fast movement plugin
- Copying and pasting with the power of registers

Technical requirements

This chapter will cover working with a Python code base. You can get the code we'll be editing in this chapter from GitHub at `https://github.com/PacktPublishing/Mastering-Vim-Second-Edition/tree/main/Chapter02`.

Installing plugins

This chapter will start by introducing Vim plugins. Plugin management is a rather broad subject (and it's covered in *Chapter 3*, *Follow the Leader – Plugin Management*, as well), but we're starting out with just a few plugins, so we won't have to worry ourselves with that topic yet.

First, let's go through the one-time setup:

1. You'll need to create a directory to store plugins. Execute the following on the command line:

```
$ mkdir -p ~/.vim/pack/plugins/start
```

> **For Windows users**
>
> If you're using GVim under Windows, you'll have to create the `vimfiles` directory under your user folder (usually `C:\Users\<username>`), and then create `pack\plugins\start` folders inside of it.

2. You'll want to tell Vim to load documentation for each plugin, as it doesn't do so automatically. For that, add the following line to your `~/.vimrc` file:

```
silent! helptags ALL  " Load help files for all plugins.
```

   Now, every time you want to add a plugin, you'll have to do the following.

3. Find your plugin on GitHub. For example, let's install `https://github.com/tpope/vim-unimpaired`, a plugin that adds a set of handy mappings (we'll get to using it later in this chapter). If you have Git installed, find the Git repository URL (in this case, it's `https://github.com/tpope/vim-unimpaired`) and run the following:

```
$ git clone https://github.com/tpope/vim-unimpaired \
  ~/.vim/pack/plugins/start/vim-unimpaired
```

> **If you don't have Git installed**
>
> If you don't have Git installed, or if you're installing a plugin for GVim under Windows, navigate to the plugin's GitHub page, and find the **Clone or download** button. Download the ZIP archive and unpack it into `.vim/pack/plugins/start/vim-unimpaired` in Linux or `vimfiles/pack/plugins/start/vim-unimpaired` in Windows.

4. Restart Vim and the plugin should be available to use.

> **How do I know if the plugin works?**
>
> You can usually check if the plugin is ready to use by invoking one of its mappings or other side effects (which you can look up, say, on GitHub). For instance, vim-unimpaired introduces *]f* mapping to move to the next file in the current directory. You can press *]* followed by *f* and see if something happens.

But before you get to use the plugin we just installed (and to understand why it's useful), let's talk about how your workspace is organized in Vim.

Organizing the workspace

So far, we've only worked with a single file in Vim. When working with code, you usually have to work with multiple files at once, switching back and forth, making edits across multiple files, and looking up certain bits somewhere else. Luckily, Vim provides an extensive way to deal with many files:

- Buffers are the way Vim internally represents files; they allow you to switch between multiple files quickly
- Windows organize the workspace by displaying multiple files next to each other
- Tabs are a collection of windows
- Folds allow you to hide and expand certain portions of files, making large files easier to navigate

Here's a screenshot illustrating the preceding points:

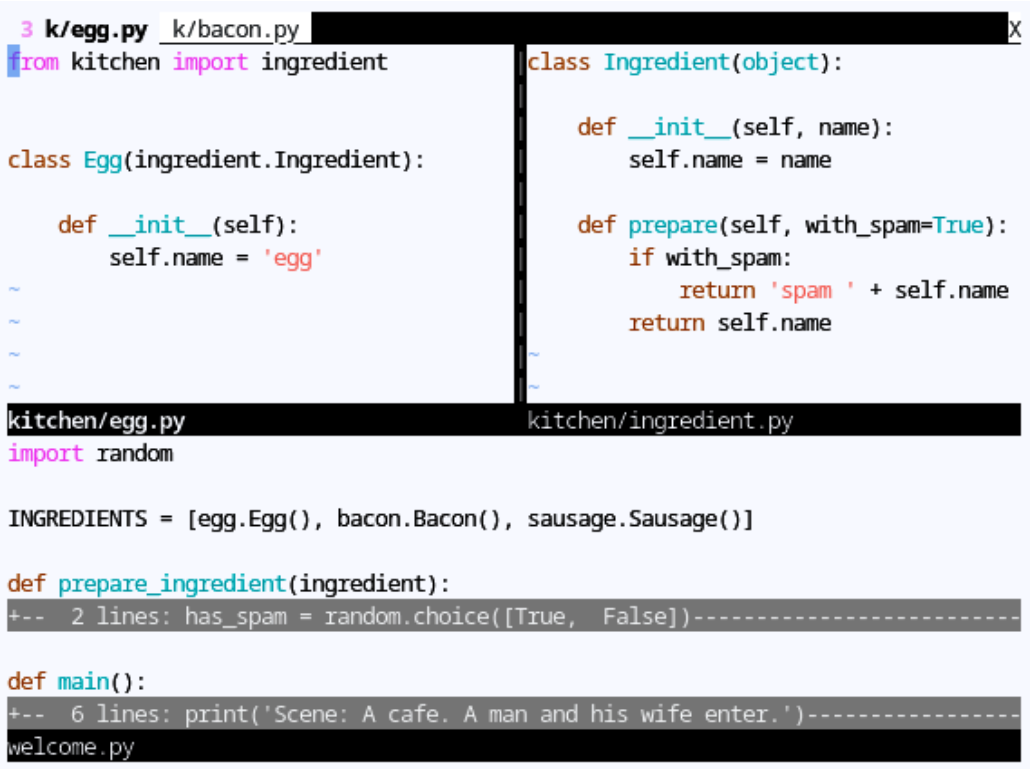

Figure 2.1 – Buffers, windows, tabs, and folds conveniently illustrated in one place.

Let's understand the content in the screenshot:

- Multiple files (labeled `kitchen/egg.py`, `kitchen/ingredient.py`, and `welcome.py`) are open as windows
- The bar at the top (listing `3 k/egg.py` and `k/bacon.py`) indicates the tabs
- Lines starting with `+--` indicate folds, hiding away portions of a file
- Every open file is loaded into a buffer, you'll learn about those in a moment

This section will go over windows, tabs, and folds in detail, and you'll be able to comfortably work with as many files as you need.

Buffers

Buffers are internal representations of files. Every file you open will have a corresponding buffer. Let's open a file from the command line: `vim welcome.py`. Now, let's see a list of existing buffers:

```
:ls
```

> **Command synonyms**
>
> Many commands have synonyms, and `:ls` is not an exception: `:buffers` and `:files` will accomplish the same thing. Pick one that's the easiest for you to remember!

Here's what the output of `:ls` looks like (see the bottom three lines):

```
    has_spam = random.choice([True,  False])
    ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for ingredient in INGREDIENTS:
        menu.append(prepare_ingredient(ingredient))
:ls
  1 %a   "welcome.py"                    line 12
Press ENTER or type command to continue
```

Figure 2.2 – You can see the output of the `:ls` command at the bottom of the screen.

The status bar shows some information about the buffers we have open (we only have one right now):

- `1` is the buffer number, and it'll stay constant throughout the Vim session
- `%` indicates that the buffer is in the current window (see the *Windows* section)

- `a` signals that the buffer is active: it's loaded and is visible
- `"welcome.py"` is the filename
- `line 12` is the current cursor position

Let's open another file:

```
:e kitchen/bacon.py
```

You can see that the file we initially opened is nowhere to be seen and has been replaced with the current file. However, `welcome.py` is still stored in one of the buffers. List all of the buffers again:

```
:ls
```

You can see both filenames listed:

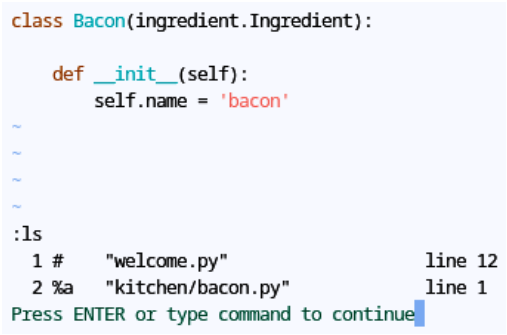

Figure 2.3 – Output of the `:ls` command, listing "welcome.py" and "kitchen/bacon.py" buffers.

How do we get to the file, then?

Vim refers to buffers by a number and a name, and both are unique within a single session (until you exit Vim). To switch to a different buffer, use the `:b` command, followed by the number of the buffer:

```
:b 1
```

You can shorten the previous by omitting the space between `:b` and the buffer number: `:b1`.

Voila, you're taken back to the original file! Since buffers are also identified by a filename, you can switch between them using partial filenames. The following will open the buffer containing `kitchen/bacon.py`:

```
:b bacon
```

However, if you have more than one match, you'll get an error. Try looking for a buffer with a filename containing `py`:

```
:b py
```

As you can see in the following screenshot, the status line displays an error:

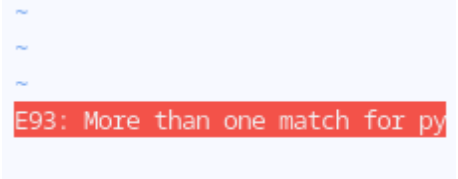

Figure 2.4 – You get an error when more than one buffers return a partial match!

That's when you can use tab completion to cycle through the available options. Type in `:b py` (without hitting *Enter*) and press the *Tab* key to cycle through the available results.

You can also cycle through buffers using `:bn` (`:bnext` for next buffer) and `:bp` (`:bprevious` for previous buffer).

Once you're done with the buffer, you can delete it, hence removing it from the list of open buffers without quitting Vim:

```
:bd
```

If you modified the contents of the buffer, this will return an error if the current buffer is not saved. Hence, you'll get a chance to save the file without accidentally deleting the buffer.

Plugin spotlight – unimpaired

Tim Pope's **vim-unimpaired** is a plugin that adds a number of handy mappings for existing Vim commands (and a few new ones). I use it daily, as I find mappings more intuitive—*]b* (alias for `:bn`) and *[b* (alias for `:bp`) cycle through open buffers, *]f* and *[f* cycle through files in a directory, and so on. It's available from GitHub at `https://github.com/tpope/vim-unimpaired`.

Installation instructions

If you haven't installed any plugins yet, see *Installing plugins* at the beginning of the chapter for one-time setup instructions. After that's been done at least once, run the following (if you're using Windows, you'll need to change the directory to `~\pack\plugins\start`) and restart Vim:

```
$ git clone https://github.com/tpope/vim-unimpaired \
  ~/.vim/pack/plugins/start/vim-unimpaired
```

Here are some of the mappings vim-unimpaired provides:

- *]b* and *[b* cycle through buffers
- *]f* and *[f* cycle through files in the same directory as the current buffer
- *]l* and *[l* cycle through the location list (see the *Location list* section in *Chapter 5, Build, Test, and Execute*)
- *]q* and *[q* cycle through the quickfix list (see the *Quickfix list* section in *Chapter 5*, *Build, Test, and Execute*)
- *]t* and *[t* cycle through tags (see the *Meet Exuberant Ctags* section in *Chapter 4*, *Understanding the Text*)

The plugin also allows you to toggle certain options with just a few key presses, such as *yos* to toggle spell checking or *yoc* to toggle the cursor line highlighting. It's cool.

See `:help unimpaired` for a full list of mappings and features that vim-unimpaired provides.

Windows

Vim loads buffers into windows. You can have multiple windows open on the screen at the same time, allowing for split-screen functionality.

Creating, deleting, and navigating windows

Let's give working with windows a shot. Open `welcome.py` (either from a command line by running `$ vim welcome.py` or from Vim with `:e welcome.py`).

Open one of our files in a split window:

```
:split kitchen/bacon.py
```

> **Type less!**
> You can shorten this command to `:sp`.

You can see that `kitchen/bacon.py` was opened above the current file and that your cursor was placed there:

Figure 2.5 – A `:split` command usually opens a new window above the current one.

You can split the window vertically as well by running the following code:

```
:vsplit kitchen/egg.py
```

As you can see, this creates another window in a vertical split (your cursor is now moved to the new window):

Figure 2.6 – A `:vsplit` command opens a new window to the left of the current one.

Shortening commands

`:vs` is a shorter version of the `:vsplit` command.

You can combine the `:split` and `:vsplit` commands indefinitely to create as many windows as you need.

All of the commands you've learned so far will function as usual within this window, including changing buffers. To move between the windows, use *Ctrl* + *w*, followed by a directional key: *h*, *j*, *k*, or *l*. Arrow keys work as well.

Faster window movement

If you use windows a lot (the feature, not the operating system), you might benefit from binding *Ctrl* + *h* to go to the split to the left, *Ctrl* + *j* to go to the split at the bottom, and so on. Add the following to your `.vimrc` file:

```
" Fast split navigation with <Ctrl> + hjkl.
noremap <c-h> <c-w><c-h>
noremap <c-j> <c-w><c-j>
noremap <c-k> <c-w><c-k>
noremap <c-l> <c-w><c-l>
```

Do note that some of those keys have default functionality which those mappings will override. For instance, *Ctrl* + *l* redraws the screen (which can be helpful if your Vim instance starts acting up – but you can manually call `:redraw` instead). In some terminal emulators, *Ctrl* + *h* is mapped to a backspace key, and *Ctrl* + *j* is mapped to the *Enter* key, which can cause unexpected behaviors.

Give it a shot: *Ctrl* + *w* followed by *j* will move you to the window below, and *Ctrl* + *w*, *k* will move the cursor back up.

You can close the split window in one of the following ways:

- *Ctrl* + *w*, followed by *q* will close the current window
- `:q` will close the window and unload the buffer; however, this will close Vim if you only have one window open
- `:bd` will delete the current buffer and close the current window
- *Ctrl* + *w*, followed by *o* (or the `:only` or `:on` command) will close all windows except for the current one

Quitting with many windows open

When you have multiple windows open, you can quit them all and exit Vim by executing `:qa`. This can be combined with the `:w` command to save every open file and quit: `:wqa`.

If you want to close a buffer without closing the window it's in, you can add the following command to your `.vimrc` file:

```
command! Bd :bp | bd #  " Close buffer without closing window.
```

You'll be able to use `:Bd` to close the buffer while keeping a split window open.

Moving windows

Windows can also be moved, swapped, and resized. Since there's no drag-and-drop functionality in Vim, there are some commands you will have to remember.

> **I lied**
>
> If your terminal emulator supports it (and most modern ones do) or you're using gVim, you can enable mouse control in Vim with `:set mouse=a`, which will allow resizing windows with a mouse. It takes me out of the editing flow, so I don't like having the mouse control enabled.

> **Remember to ask for help!**
>
> You don't have to remember all of these commands, as long as you know what window operations are supported. `:help window-moving` and `:help window-resize` will take you to the corresponding entries in the Vim manual when you inevitably forget the shortcuts.

As with the rest of the window commands, these are prefixed by *Ctrl* + *w*.

Ctrl + *w* followed, by an uppercase movement key (*H*, *J*, *K*, or *L*) will move the current window to the corresponding position:

- *Ctrl* + *w*, *H* moves the current window to the leftmost part of the screen
- *Ctrl* + *w*, *J* moves the current window to the bottom of the screen
- *Ctrl* + *w*, *K* moves the current window to the top of the screen
- *Ctrl* + *w*, *L* moves the current window to the rightmost part of the screen

For example, let's start with the following window layout (which was achieved by opening `welcome.py` and running `:sp kitchen/bacon.py`, followed by `:vs kitchen/egg.py`):

Figure 2.7 – Files `welcome.py`, `kitchen/egg.py`, `kitchen/bacon.py` open in split windows.

Note the cursor position (in `kitchen/egg.py`). Here's what happens when we try to move the window containing the `kitchen/egg.py` buffer in each direction:

- *Ctrl* + *w*, *H* migrates `kitchen/egg.py` all the way to the left:

Figure 2.8 – *Ctrl* + *w*, *H* migrates `kitchen/egg.py` all the way to the left.

- *Ctrl* + *w*, *J* moves `kitchen/egg.py` to the bottom of the screen, turning a vertical split into a horizontal split:

Figure 2.9 – *Ctrl* + *w*, *J* moves `kitchen/egg.py` to the bottom of the screen, turning a vertical split into a horizontal split.

- *Ctrl* + *w*, *K* moves `kitchen/egg.py` to the top of the screen:

Figure 2.10 – *Ctrl* + *w*, *K* moves `kitchen/egg.py` to the top of the screen.

- *Ctrl* + *w*, *L* moves `kitchen/egg.py` to the right of the screen:

Figure 2.11 – *Ctrl* + *w*, *L* moves `kitchen/egg.py` to the right of the screen.

You can change the contents of each window by simply navigating to it and selecting the desired buffer using the `:b` command. There are, however, options for swapping window contents:

- *Ctrl* + *w*, *r* moves every window within the row or the column (whichever is available—rows are given preference over columns) to the right or downward. *Ctrl* + *w*, *R* performs the same operation in reverse.
- *Ctrl* + *w*, *x* exchanges the contents of a window with the next one (or a previous one if it's considered a last window).

Window numbering

Internally, Vim refers to windows by number unique within the current tab, and by a unique ID that functions across tabs. Some window management commands take the window number or an ID as an argument, but this book will not be covering these. You can read `:help winid` for more information.

Resizing windows

Default 50/50 window proportions might not be exactly what you're looking for, and there are some options for changing sizes.

Ctrl + *w* followed by = will equalize the height and width of all open windows. This is really useful when you just resized the Vim window and the height of your windows got all messed up.

The `:resize` command increases or decreases the height of a current window, while `:vertical resize` will adjust the width of the window. You can use these as follows:

- `:resize +N` will increase the height of a current window by `N` rows
- `:resize -N` will decrease the height of a current window by `N` rows
- `:vertical resize +N` will increase the width of a current window by `N` columns
- `:vertical resize -N` will decrease the width of a current window by `N` columns

Shortening commands

`:resize` and `:vertical resize` can be shortened to `:res` and `:vert res`. There are also keyboard shortcuts for changing the height and width by one: *Ctrl* + *w*, - and *Ctrl* + *w*, + adjust the height, while *Ctrl* + *w*, > and *Ctrl* + *w*, < adjust the width.

Command modifiers

`:vertical` or `:vert` is a command modifier that can be applied to many other commands: for instance, the `:vert split` command will produce the same result as `:vsplit`.

Both commands can also be used to set the height and the width to a specific number of rows or columns:

- `:resize N` will set the height of the window to `N`
- `:vertical resize N` will set the width of the window to `N`

Tabs

In many modern editors, tabs are used to represent different files. While you can certainly do this in Vim, you might want to consider their original purpose.

Vim uses tabs to switch between collections of windows, allowing you to effectively have multiple workspaces. Tabs are often used to work on a slightly different problem or set of files within the same Vim session. Personally, I don't get a lot of use out of tab pages, but if you find yourself often switching context within the project or between projects, then tabs might be exactly what you're looking for.

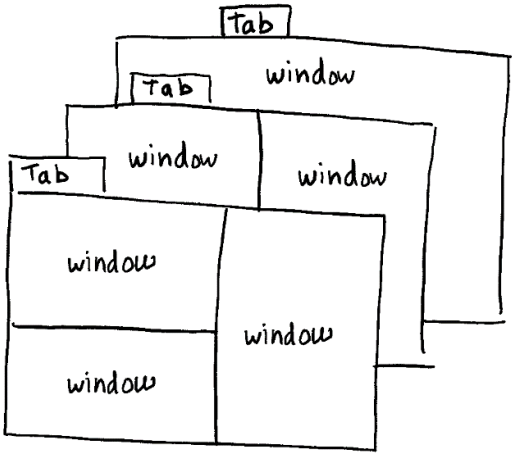

Figure 2.12 – Tabs contain collections of windows (the concept is often reversed in modern editors).

You can open a new tab with an empty buffer as follows:

```
:tabnew
```

> **Tip**
> You can open an existing file in a new tab by running `:tabnew <filename>` or `:tabedit <filename>`.

As you can see, tabs are displayed on the top of the screen. The tab labeled `3 welcome.py` is a tab with three open windows and an active buffer, `welcome.py`. The **[No Name]** tab is the one we just opened:

Figure 2.13 – `:tabnew` command open a new tab.

You can load a file in it in the usual way: `:e <filename>`. You can also switch to a desired buffer using the `:b` command.

To navigate between tabs, you can use the following:

- *gt* or `:tabnext` to move to the next tab
- *gT* or `:tabprevious` to move to the previous tab

The tabs can be closed using `:tabclose` or by closing all of the windows it contains (for example, with `:q` if it's the only window).

`:tabmove N` lets you place the tab after the *N*th tab (or as a first tab if `N` is `0`).

Folds

One of the most powerful tools Vim provides for navigating large files is folds. Folds allow you to hide portions of the file, either based on some predefined rules or manual fold markers.

This is how `welcome.py` looks with some sections folded:

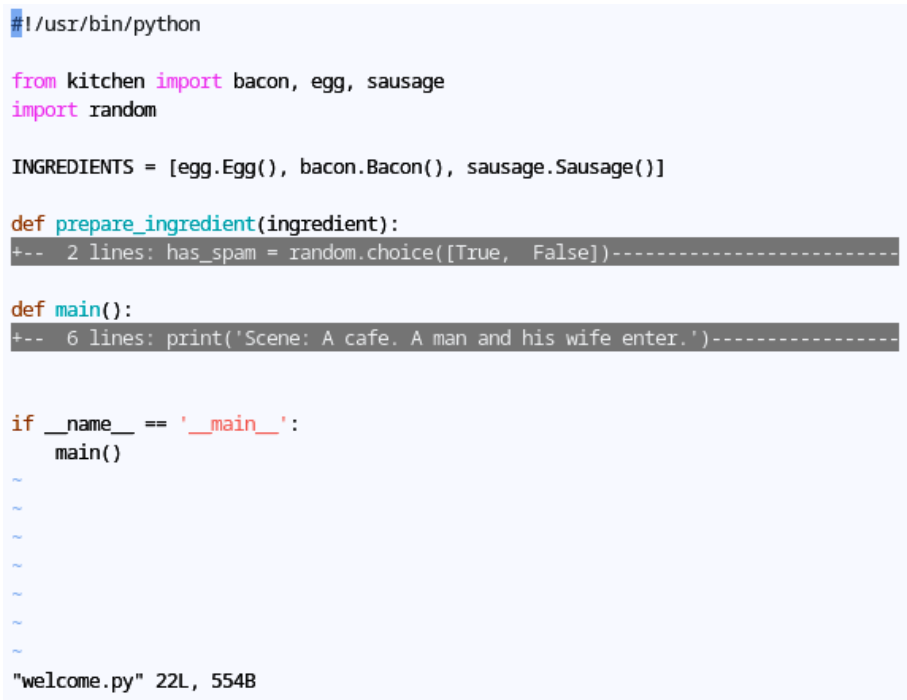

Figure 2.14 – `prepare_ingredient` and `main` method code is folded.

Method content is hidden, allowing you to view the code from a bird's-eye view.

Folding Python code

Since we're working with Python code throughout this book, let's play with some folds in our code. First, you'll need to change a setting called `foldmethod` to `indent` in your `.vimrc` file. Let's only do this for Python file by prefixing the `set` command with `autocmd filetype python`:

```
autocmd filetype python set foldmethod=indent
```

> **Reload!**
>
> Don't forget to reload your `~/.vimrc` file by either restarting Vim or executing `:source $MYVIMRC`.

This will tell Vim to fold based on indentation (there are multiple ways to work with folds; see the following section, *Types of folds*, for more information).

Open `welcome.py` and you will see portions of our file folded away:

```
#!/usr/bin/python

from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
+--  2 lines: has_spam = random.choice([True,  False])-------------------------

def main():
+--  6 lines: print('Scene: A cafe. A man and his wife enter.')----------------


if __name__ == '__main__':
    main()
~
~
~
~
~
~
~
"welcome.py" 22L, 554B
```

Figure 2.15 – With `foldmethod=indent`, Python method code is folded.

Navigate your cursor to one of the folded lines. Hitting *zo* will open the current fold:

Figure 2.16 – *zo* opens the fold under cursor and *zc* closes it.

Whenever your cursor is over a potential fold (an indented chunk of code in this example), *zc* will close the fold.

Visualizing folds

To visualize where folds are, you can use `:set foldcolumn=N`, where N is 0..12. This will dedicate the first N columns to the left of the screen to indicate folds with the - (beginning of an open fold), | (contents of an open fold), and + (closed fold) symbols.

You can also use *za* to toggle folds (open closed folds and close open folds).

You can open and close all folds in the file at the same time using *zR* and *zM*, respectively.

Tip

Setting an automatic `foldmethod` setting (such as `indent`) will display all files as folded by default. It's a matter of preference, and you may prefer to have the folds open when opening a new file. Adding `autocmd BufRead * normal zR` to your `.vimrc` file will keep the folds open as you open new files. This command tells Vim to execute `zR` (open all folds) when reading a buffer.

Alternatively, you can `set foldlevelstart=5` (or higher) in your `.vimrc` to have folds up to a fifth level open by default.

Types of folds

Vim is somewhat intelligent when it comes to folding code, and supports multiple ways to fold code. The folding method is guarded by the `foldmethod` option in your `.vimrc` file. Supported fold methods are as follows:

- `manual` allows you to manually define folds. This becomes unrealistic when working with any substantial body of text.
- `indent` supports indentation-based folding, which is perfect for languages and code bases where indentation matters (regardless of the language you're working with, a standardized code base is likely to have some consistent indentation, making `indent` a quick and easy way to fold away bits you don't care for).
- `expr` allows using Vim scripting language to define folds. This is an extremely powerful tool if you have complex custom rules you'd like to use for defining folds.
- `marker` uses special markup in the text, such as { { { and } } }. This is useful for managing long `.vimrc` files but has little use outside of the Vim world since it requires modifying file content.
- `syntax` provides syntax-aware folds. However, not every language is supported out of the box (Python isn't).
- `diff` is automatically used when Vim operates in a diff mode, displaying the difference between two files (see *Vimdiff* in *Chapter 5*, *Build, Test, and Execute*).

> **Options in `.vimrc`**
>
> Reminder: you can set an option in your `.vimrc` file by adding the following line: `autocmd filetype <filetype> set foldmethod=<method>`.

Now that you have an idea of how to organize the files in front of you, let's look at how to navigate complex file trees.

Navigating file trees

Since software projects contain a lot of files and directories, finding a way to traverse and display these using Vim comes in handy. This section will cover five different ways that you can navigate your files: using the built-in Netrw file manager or using the `:e` command with the `wildmenu` option enabled, as well as using the NERDTree, Vinegar, and CtrlP plugins. All of these provide different ways to interact with files and can be mixed and matched.

Netrw

Netrw is a built-in file manager in Vim (if we want to get technical, it's a plugin that ships with Vim). It allows you to browse directories and functions, similar to any other file manager you've worked with in your favorite OS.

Use `:Ex` (the full command is `:Explore`) to open the file navigation window:

Figure 2.17 – Netrw file manager window, with a handy help screen above the directory tree.

Netrw is native to Vim

Netrw is fully integrated with Vim, so executing an edit command on a directory (for example, `:e .` to open a current directory) actually opens Netrw. This also makes it so that you have one less command to remember.

Here, you can see all of the files within our workspace. Netrw already provides a quick help section in a status bar, but here are the main controls you'll need to know:

- *Enter* opens files and directories
- - goes up a directory
- *D* deletes a file or directory
- *R* renames a file or directory

A Netrw window can be opened in split windows or new tabs as well:

- `:Vex` opens Netrw in a vertical split
- `:Sex`, despite what the name might imply, opens Netrw in a horizontal split
- `:Lex` opens Netrw in a leftmost full-height vertical split

Netrw is a powerful tool, which supports remote editing as well; for instance, to get a directory listing over SFTP, you can run the following:

```
:Ex sftp://<domain>/<directory>/
```

You can substitute `:Ex` with `:e` for the same results. You can edit individual files as well. Here's how to open a file over SCP:

```
:e scp://<domain>/<directory>/<file>
```

:e with wildmenu enabled

Another way to explore file trees is to use the `set wildmenu` option in your `.vimrc` file. This option sets an autocomplete menu to operate in enhanced mode, showing possible autocomplete options above the status line. With `wildmenu` enabled, type in `:e` (followed by a space) and hit *Tab*. This will bring up a list of files in the current directory, and you can use the *Tab* key to iterate through these and *Shift* + *Tab* to move backward (the left and right arrow keys perform the same function):

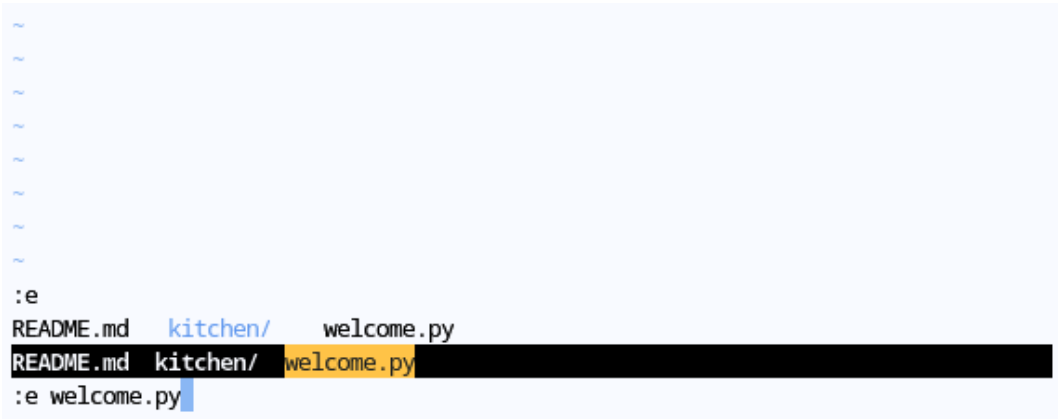

Figure 2.18 – With the `set wildmenu` option, you can cycle through commands by pressing *Tab*.

Pressing *Enter* will open the selected file or directory. The down arrow allows you to drill down into directory under cursor, and the up arrow takes you back up a level.

This also works with partial paths, and entering `:e <beginning_of_filename>` followed by the *Tab* key invokes `wildmenu` as well.

My `.vimrc` file has the following in it:

```
set wildmenu " Enable enhanced tab autocomplete.
set wildmode=list:longest,full " Complete till longest string,
" then open the wildmenu.
```

> **New in Vim 9**
>
> In Vim 9, you can also use `set wildoptions=pum` to display the `wildmenu` as a vertical popup window – that's pretty cool!

This allows you to autocomplete the path to the longest match possible (and display a list of possible completion options) on the first *Tab* press, and traverse through files with `wildmenu` on the second *Tab* press.

Plugin spotlight – NERDTree

NERDTree is a handy plugin that emulates modern IDE behavior by displaying a file tree in a split buffer to the side of the screen. NERDTree is available from `https://github.com/scrooloose/nerdtree`.

> **Installation instructions**
>
> If you haven't installed any plugins yet, see *Installing plugins* at the beginning of the chapter for one-time setup instructions. After that's been done at least once, run the following (if you're using Windows, you'll need to change the directory to `~\pack\plugins\start`) and restart Vim:
>
> ```
> $ git clone https://github.com/scrooloose/nerdtree \
>   ~/.vim/pack/plugins/start/nerdtree
> ```

Once installed, you can invoke NERDTree by typing the following:

```
:NERDTree
```

A list of files in a directory will show up:

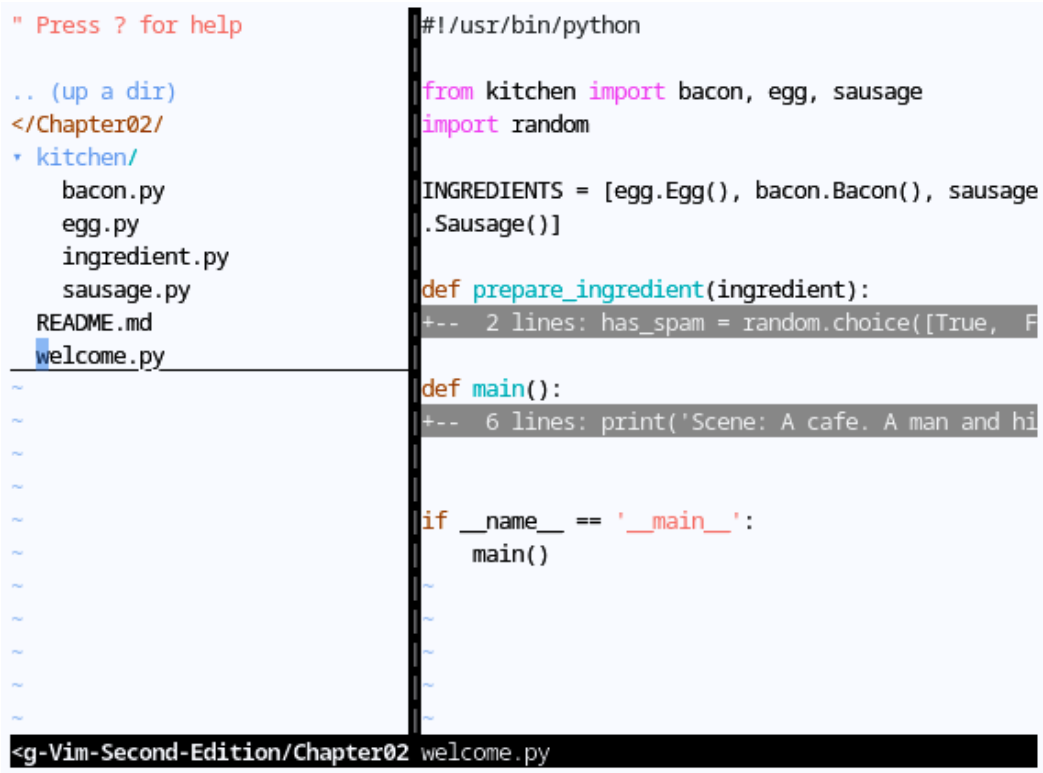

Figure 2.19 – NERDTree file manager window, an alternative to using native Netrw.

Use *h*, *j*, *k*, and *l* or the arrow keys to navigate the file structure, and *Enter* or *o* to open the file. There are multiple useful shortcuts, and *Shift* + *?* brings up a handy cheat sheet.

A notable feature is bookmark support, which allows you to bookmark a directory (when placing the cursor over it in NERDTree) by executing `:Bookmark`. Press *B* when in a NERDTree window to display bookmarks at the top of the window.

In the following screenshot, you can see the bookmarks I have for code that supports chapters of this book (the `Chapter01/` and `Chapter02/` directories):

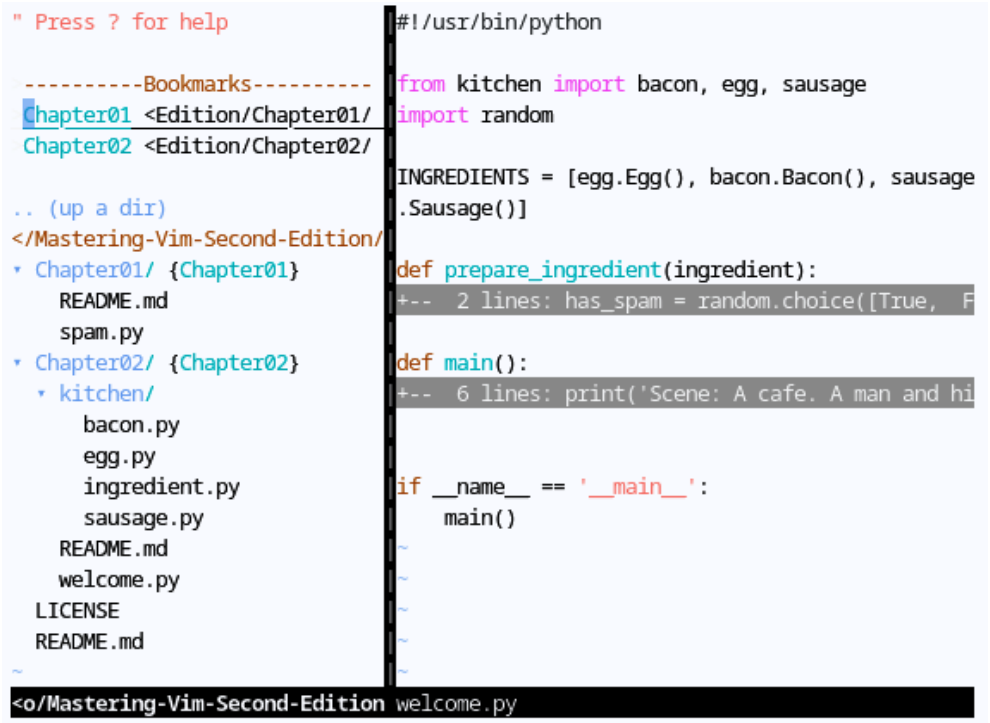

Figure 2.20 – You can see NERDTree bookmarks to `Chapter01/` and `Chapter02/` in the top-left corner of the screen.

You can choose to always display bookmarks in a NERDTree window by setting the `NERDTreeShowBookmarks` option in your `.vimrc` file:

```
let NERDTreeShowBookmarks=1 " Display bookmarks on startup.
```

You can bring NERDTree up or hide it by executing `:NERDTreeToggle`. If you're interested in having NERDTree up every time you're editing, you might want to add the following to your `.vimrc` file:

```
autocmd VimEnter * NERDTree " Enable NERDTree on Vim startup.
```

Something I personally found really useful is to close the NERDTree window automatically when it's the last open window. I have the following in my `.vimrc` file:

```
" Autoclose NERDTree if it's the only open window left.
autocmd bufenter * if (winnr("$") == 1 && exists("b:NERDTree") &&
  \ b:NERDTree.isTabTree()) | q | endif
```

These days, I rarely find myself using NERDTree. Before switching to Vim, I often relied on having a project outline in the view when I worked. In my early days of learning Vim, NERDTree was a lifesaver. Vim changed the way I work and having a file outline always on became distracting, so eventually I moved back to using Netrw.

Plugin spotlight – Vinegar

Tim Pope's `vinegar.vim` is a simple plugin that addresses the difficulty of using project drawers with Vim split window functionality. Plugins such as NERDTree become rather disorienting to work with when you have more than one window open in a split.

In the following example, three windows are open (and a fourth NERDTree window on the left):

Figure 2.21 – NERDTree and split windows are sometimes hard to manage. Where do you think the next file will open?

When pressing *Enter* with the cursor in the NERDTree window, which window will the new file be opened in?

Hint: it's the bottom-left one—but you don't have a way of knowing that. NERDTree opens files in the last created window.

Tim Pope solves this problem with a small plugin called **Vinegar**, which makes using Netrw a more seamless experience. It's available over at `https://github.com/tpope/vim-vinegar`.

Installation instructions

If you haven't installed any plugins yet, see *Installing plugins* at the beginning of the chapter for one-time setup instructions. After that's been done at least once, run the following (if you're using Windows, you'll need to change the directory to `~\pack\plugins\start`) and restart Vim:

```
$ git clone https://github.com/tpope/vim-vinegar \
  ~/.vim/pack/plugins/start/vim-vinegar
```

NERDTree and Netrw integration

If you have NERDTree installed when using Vinegar, you'll get a NERDTree window instead of Netrw. To avoid NERDTree replacing Netrw (and to make commands like - work), set `let NERDTreeHijackNetrw=0` in your `.vimrc` file above the `packloadall` command.

Vinegar adds a handy new mapping: `-` (dash) to open Netrw in a current directory. Try it out:

Figure 2.22 – The Netrw file manager accessed through vinegar.vim.

The plugin hides Netrw's help bar, which might be confusing at first. Hit *I* (uppercase *i*) to bring it back. Another shortcut, the `~` key, takes you to your home `directory`, which is often where you'll store your projects.

Plugin spotlight – CtrlP

CtrlP is a fuzzy completion plugin that helps you open the files you need quickly, given that you somewhat know what you're looking for. CtrlP is available from `https://github.com/ctrlpvim/ctrlp.vim`.

Installation instructions

If you haven't installed any plugins yet, see *Installing plugins* at the beginning of the chapter for one-time setup instructions. After that's been done at least once, run the following (if you're using Windows, you'll need to change the directory to `~\pack\plugins\start`) and restart Vim:

```
$ git clone https://github.com/ctrlpvim/ctrlp.vim \
  ~/.vim/pack/plugins/start/ctrlp.vim
```

Install it and hit *Ctrl* + *p*:

Figure 2.23 – Fuzzy filename completion provided by CtrlP.

This shows the list of files in the project directory. Type in a partial filename or a path, and the list of files will narrow down to string matches. You can use *Ctrl* + *j* and *Ctrl* + *k* to navigate up and down the list and *Enter* to open the file. *Esc* closes the CtrlP window.

CtrlP also allows you to navigate through buffers and the most recently used files. With the CtrlP window open, you can use *Ctrl* + *f* and *Ctrl* + *b* to cycle through the available options.

You can invoke these directly by executing `:CtrlPBuffer` for buffers and `:CtrlPMRU` for the most recently used ones. You can also use `:CtrlPMixed` to search through files, buffers, and the most recently used files at the same time.

You can also add custom mappings for these to your `.vimrc` file. For example, to map *Ctrl* + *b* to `:CtrlPBuffer`, you could do the following:

```
nnoremap <C-b> :CtrlPBuffer<cr> " Map CtrlP buffer mode to Ctrl + b.
```

CtrlP excels at moving between files, but when it comes to navigating the text within a file, Vim has some powerful built-in tools.

Navigating text

We've covered some basic movements (by characters, words, and paragraphs), but Vim supports a lot more options for navigation.

Check the following if you want some movement within the current line:

- As you already know, *h* and *l* move the cursor left and right, respectively
- *t* (un**t**il) followed by a character allows you to search the line for that character and place the cursor before the character, while *T* allows you to search backward
- *f* (**fi**nd) followed by a character allows you to search the current line for that character and move the cursor to the character, while *F* allows you to search backward
- _ takes you to the first non-blank character of the line, ^ takes you the beginning of the line, and *$* takes you to the end of the line

> **Text objects**
> A word consists of numbers, letters, and underscores. A WORD consists of any characters except for whitespace (like spaces, tabs, or newlines). This distinction helps with more precise navigation. For instance, `ingredient.prepare(with_spam=True)` is a single WORD, while `ingredient`, `prepare`, and `with_spam=True` are individual words.

For free-form movement, you're already familiar with these bits:

- *j* and *l* move the cursor down and up, respectively
- *w* moves you to the beginning of the next word (*W* for WORD)
- *b* moves you to the beginning of the previous word (*B* for WORD)
- *e* moves you to the end of the next word (*E* for WORD)
- *ge* moves you to the end of the previous word (*gE* for WORD)
- *Shift* + *{* and *Shift* + *}* takes you to the beginning and the end of a paragraph

Here are some new free-form movement options:

- *Shift* + *(* and *Shift* + *)* takes you to the beginning and the end of a sentence
- *H* takes you to the top of the current window, *L* takes you to the bottom of the current window, and *M* takes you to the middle of the current window
- *Ctrl* + *f* (or the *Page Down* key) scrolls the buffer one page down, and *Ctrl* + *b* (or the *Page Up* key) scrolls one page up
- */* followed by a string searches the document for a string and *Shift* + *?* to search backward

- *gg* takes you to the top of the file
- *G* takes you to the bottom of the file

This handy visualization is based on the Vim movement cheat sheet Ted Naleid published on his blog sometime in 2010 (I haven't been able to find the original source):

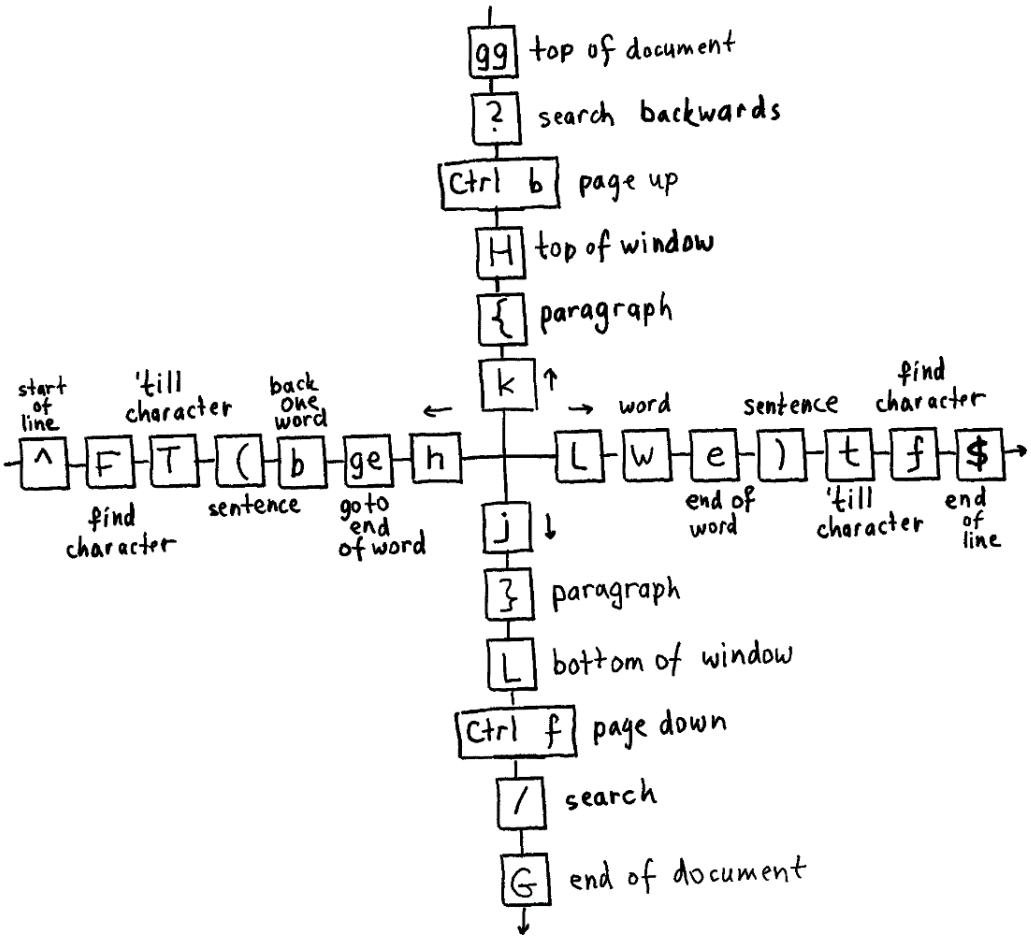

Figure 2.24 – Vim movement cheatsheet, original representation credited to Ted Naleid.

You can also move by line numbers. To enable line number display, run `:set nu`, followed by *Enter* (or add `set number` to your `.vimrc` file). Vim will dedicate a few columns on the left of the screen to display line numbers:

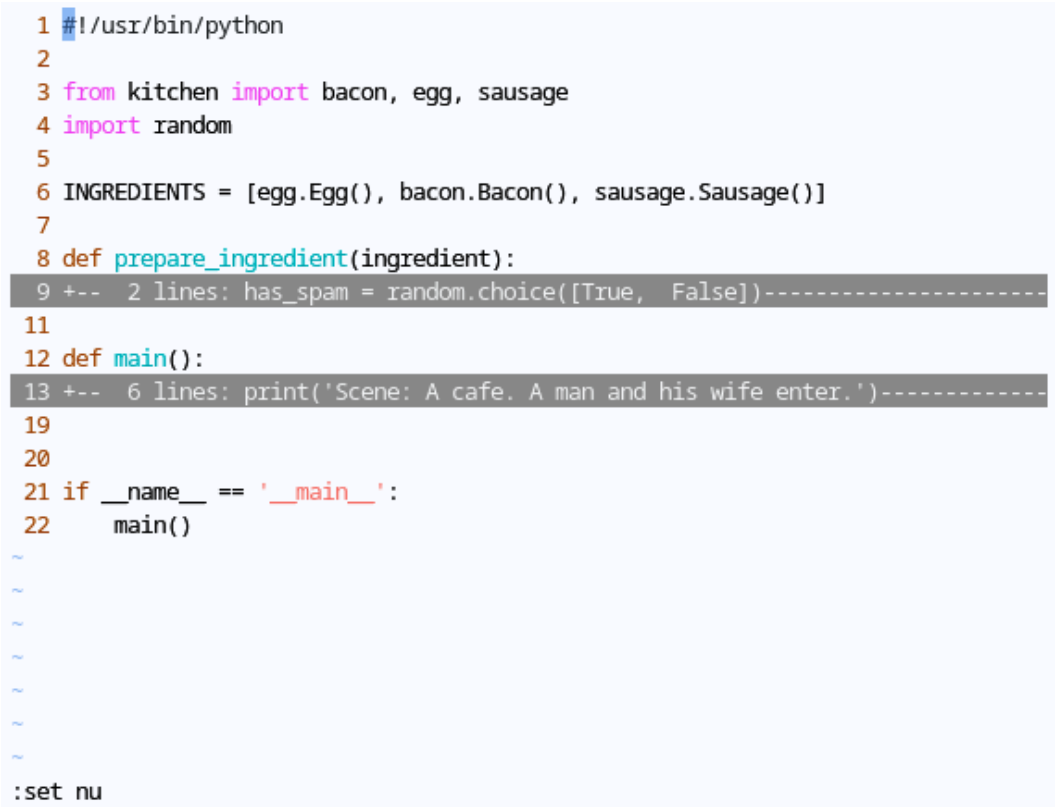

Figure 2.25 – `:set number` turns on line number display, which helps with navigation.

You can jump to a specific line by typing `:N` followed by *Enter*, where N is the absolute line number. For instance, to jump to *line 20*, you'll run `:20` followed by *Enter*.

You can also tell Vim to open a file and immediately place a cursor at a particular line. For that, add `+N` after the filename when invoking Vim, where N is the line number. For example, to open `welcome.py` on *line 14*, you'd run `$ vim welcome.py +14`.

Vim also supports relative line movement. To move down N lines you'll run `:+N` and to move down you'll run `:-N`. You can also ask Vim to display line numbers relative to the current cursor position with `:set relativenumber`. In the following screenshot, our cursor is on *line 6*, and Vim displays the relative distance to other lines:

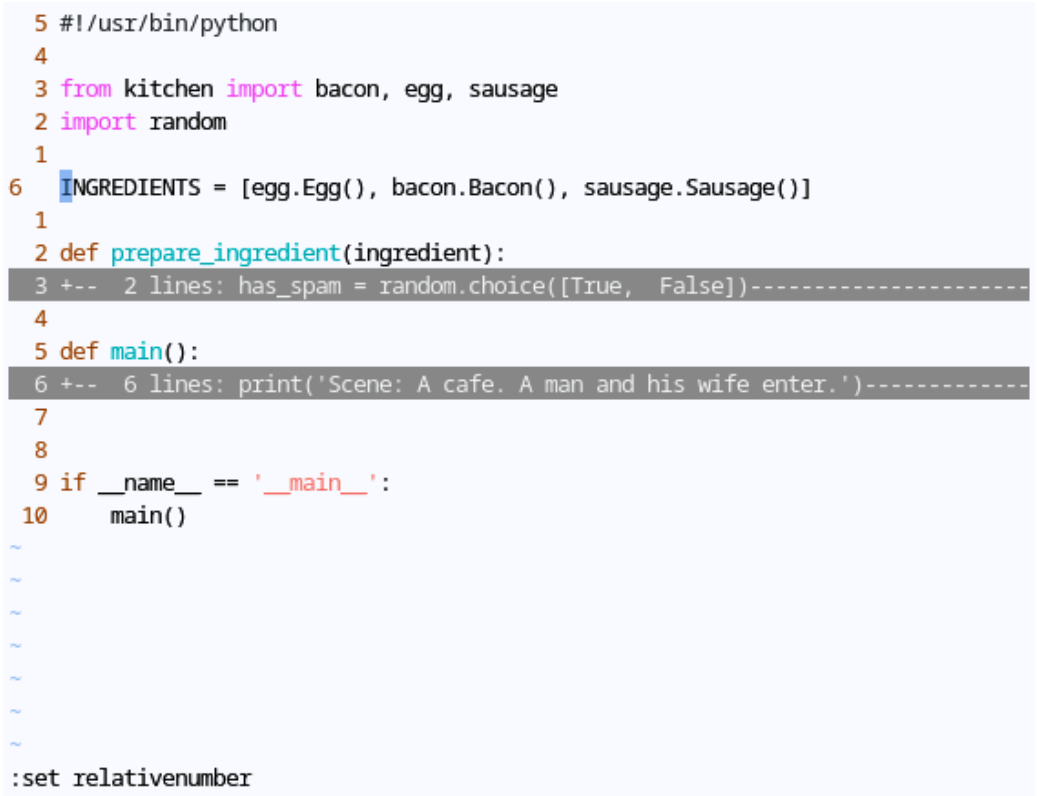

Figure 2.26 – `:set relativenumber` changes line numbers to be relative to the cursor position. To be honest, I always find that a bit disorienting.

For example, you could tell Vim to move to the line containing `def main():` by typing `:+5`, followed by *Enter*.

Jumping into insert mode

You've already learned to enter insert mode using *i*, which puts you in insert mode at the position of the cursor.

There are a few more convenient shortcuts for entering insert mode:

- *a* places you in insert mode after the cursor
- *A* places you in insert mode at the end of the line (equivalent of *$a*)
- *I* (capital *i*) places you in insert mode at the beginning of the line, but after indentation (equivalent of *_i*)
- *o* adds a new line below the cursor before entering insert mode
- *O* adds a new line above the cursor before entering insert mode
- *gi* places you in insert mode where you last exited it

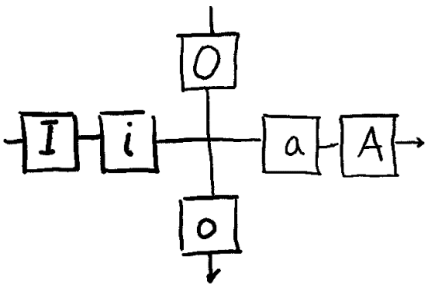

Figure 2.27 – Insert mode cheat sheet.

You've also learned how to enter insert mode after deleting some code with the change command (*c*). Here are more ways to chain change commands:

- *C* deletes text to the right of the cursor (until the end of the line) before entering insert mode
- *cc* or *S* deletes the contents of the line before entering insert mode, while preserving indentation
- *s* deletes a single character (prefix by a number to delete multiple) before placing you in insert mode

Searching with / and ?

Most times, one of the fastest ways to navigate somewhere is to search for a particular string. Vim allows you to search for a match by typing in / (which puts you in command-line mode) followed by a search string. Once you hit *Enter*, your cursor will move to the first match.

Cycling through the matches in the same buffer can be done by pressing *n* to go to the next match and *N* to the previous match.

A useful option for searching is `set hlsearch` (consider setting it in your `.vimrc` file), since it highlights every match on the screen. For example, this is how looking for `/print` in `welcome.py` looks with `hlsearch` enabled:

```
#!/usr/bin/python

from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for ingredient in INGREDIENTS:
        menu.append(prepare_ingredient(ingredient))
    print('Waitress: Well, there\'s', ', '.join(menu))


if __name__ == '__main__':
    main()
~
/print
```

Figure 2.28 – `:set hlsearch` highlights all the visible search results.

You can clear the highlights by executing `:noh`.

> **Tip**
>
> Another nifty trick is using `set incsearch`. This will make Vim dynamically move you to the first match as soon as you type.

In case you want to search backward, replace `/` with `?`. This will also affect the way `n` and `N` will behave, showing a next backward match, and a previous backward match respectively.

Searching across files

Vim has two commands to help you search across files, `:grep` and `:vimgrep`:

- `:grep` uses system `grep` and is a great tool if you're already familiar with how `grep` works
- `:vimgrep` is a part of Vim and might be easier to use if you are not already familiar with `grep`

We'll focus on `:vimgrep`, since the `grep` tool is outside the scope of this book.

The syntax is as follows: `:vimgrep <pattern> <path>`. `pattern` could either be a string or a Vim-flavored regular expression. `path` will often be a wildcard; use `**` as a path to search recursively (or `**/*.py` to restrict by file type).

Let's try searching for an `ingredient` substring in our code base:

```
:vimgrep ingredient **/*.py
```

This will take us to the first match, displaying the number of matches at the bottom of the screen:

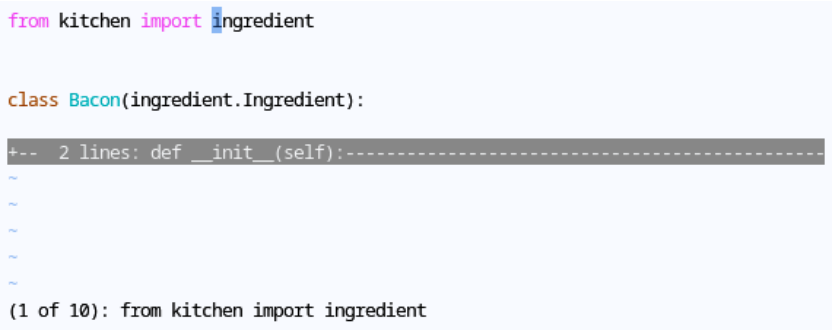

Figure 2.29 – `:vimgrep` allows you to search through files.

To navigate through the matches, use `:cn` or `:cp`. However, you might want to open a visual quickfix window by using `:copen`, as follows:

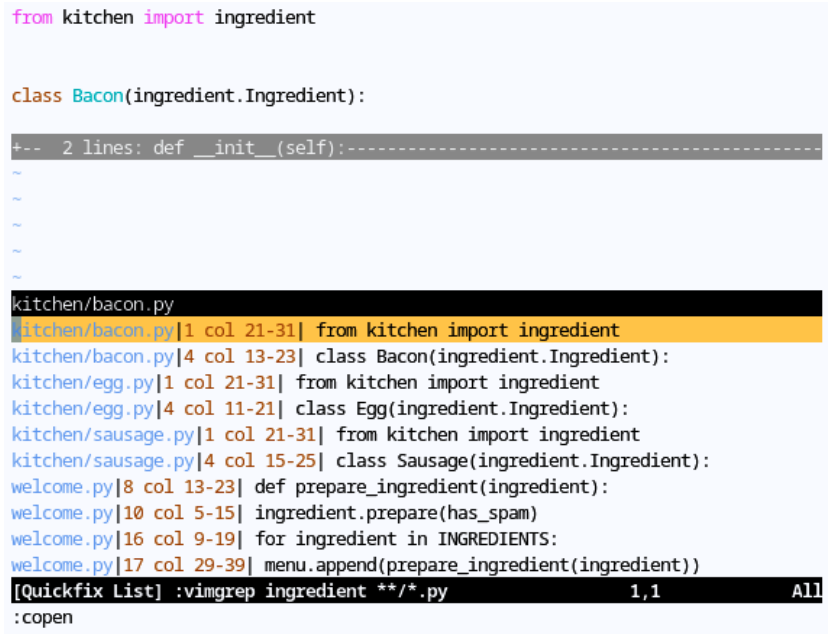

Figure 2.30 – `:copen` allows you to navigate through the visual quickfix list.

You can navigate the quickfix list with the *j* and *k* keys and jump to a match by pressing *Enter*. The quickfix window can be closed like any other window by typing `:q` or running *Ctrl* + *w*, *q*. You can read more about it in the *Quickfix list* section in *Chapter 5*, *Build, Test, and Execute*.

ack

On Linux, you can use Vim in conjunction with `ack` to search through code bases. `ack` is the spiritual successor of `grep` and is focused on working with code. Install it using your favorite package manager; here's an example of using `apt`:

```
$ sudo apt install ack-grep
```

> **Official site**
>
> Visit `https://beyondgrep.com/install` to learn more about `ack` and for installation instructions.

For example, you can now use `ack` from the command line to search for all Python files recursively (starting in the current directory) containing the word `ingredient`:

```
$ ack --python ingredient
```

The preceding code will produce output similar to `grep`:

```
~/Mastering-Vim-Second-Edition/Chapter02$ ack --python ingredient
kitchen/egg.py
1:from kitchen import ingredient
4:class Egg(ingredient.Ingredient):

kitchen/bacon.py
1:from kitchen import ingredient
4:class Bacon(ingredient.Ingredient):

kitchen/sausage.py
1:from kitchen import ingredient
4:class Sausage(ingredient.Ingredient):

welcome.py
8:def prepare_ingredient(ingredient):
10:    return ingredient.prepare(has_spam)
16:    for ingredient in INGREDIENTS:
17:        menu.append(prepare_ingredient(ingredient))
~/Mastering-Vim-Second-Edition/Chapter02$
```

Figure 2.31 – `ack` is just like `grep`, but made for working with programming languages. I find the syntax easier to remember.

Vim has a plugin that integrates the result of `ack` in Vim's quickfix window (see the *Quickfix list* section in *Chapter 5*, *Build, Test, and Execute*, to learn more about quickfix). The plugin is available from `https://github.com/mileszs/ack.vim`.

> **Installation instructions**
>
> If you haven't installed any plugins yet, see *Installing plugins* at the beginning of the chapter for one-time setup instructions. After that's been done at least once, run the following (if you're using Windows, you'll need to change the directory to `~\pack\plugins\start`) and restart Vim:
>
> ```
> $ git clone https://github.com/mileszs/ack.vim \
>   ~/.vim/pack/plugins/start/ack.vim
> ```

After installation, you will be able to execute `:Ack` from Vim:

```
:Ack --python ingredient
```

This will run `ack` and populate the quickfix window (see the preceding section, as well as *Quickfix list* in *Chapter 5*, *Build, Test, and Execute*, for more information about a quickfix window) with the output:

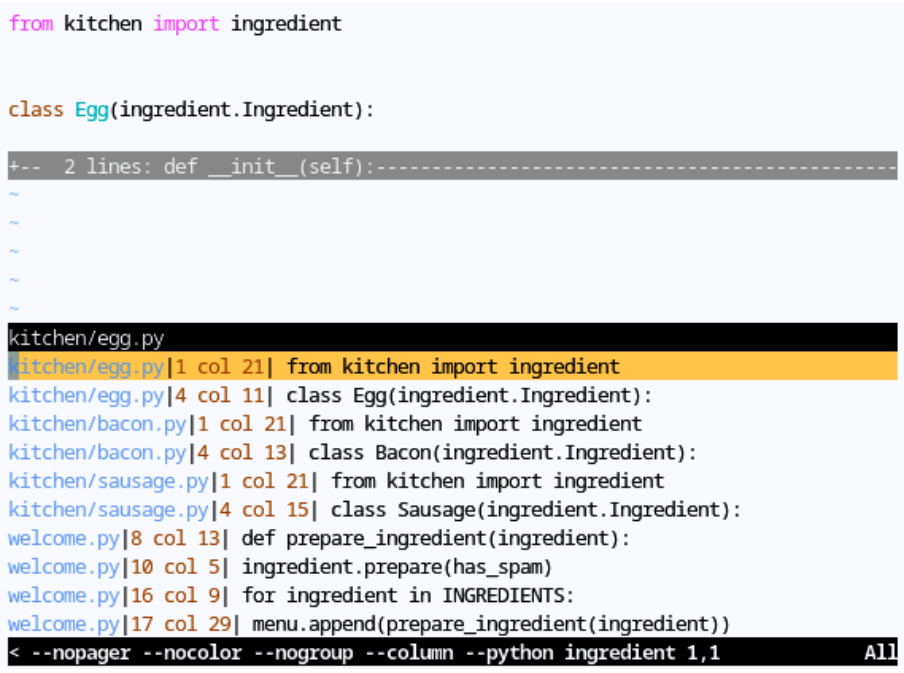

Figure 2.32 – `:Ack` searches through files and automatically opens the quickfix list.

Utilizing text objects

Text objects are an additional type of object in Vim. Text objects allow you to manipulate text within parentheses or quotes, which becomes really useful when working with code. Text objects are only available when combined with other operators like change or delete or a visual mode (see *Visual and select modes* in *Chapter 3*, *Follow the Leader – Plugin Management*).

Let's give it a shot. Navigate your cursor to the text between parentheses:

```
def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    ingredient.prepare(has_spam)
```

Now, type in *di)* (delete inside parentheses). This will delete the text inside parentheses:

```
def prepare_ingredient(ingredient):
    has_spam = random.choice()
    ingredient.prepare(has_spam)
```

This works similarly with a change command. Undo the previous change (*u*) and start in a different spot:

```
def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
```

Execute *c2aw* (change the outside of two **w**ords) to delete two words (with the surrounding spaces) and enter insert mode:

```
def main():
    print('Scene: A cafe. A man and enter.')
    print('Man: Well, what\'ve you got?')
```

Text objects come in two flavors—inner objects (prefixed by *i*) and outer objects (prefixed by *a*). Inner objects do not include white space (or other surrounding characters), while outer objects do.

A full list of text objects can be looked up through `:help text-objects`, but some interesting ones are as follows:

- `w` and `W` for words and WORDs
- `s` for sentences
- `p` for paragraphs
- `t` for HTML/XML tags

Pairs of characters that are most often used in programming are all supported as text objects: `, ', ",),], >, and } select the text enclosed by the characters.

One way to think about working with text objects is that it's like constructing sentences. Here are the two examples that we used previously broken down:

Verb	(Number)	Adjective	Noun
d delete		i inside	) parentheses
c change	2	a outside	w word

Plugin spotlight – EasyMotion

EasyMotion has been an essential part of my kit since I came across it. It simplifies navigation by allowing you to jump to the desired position with speed and precision. It's available from `https://github.com/easymotion/vim-easymotion`.

> **Installation instructions**
>
> If you haven't installed any plugins yet, see *Installing plugins* at the beginning of the chapter for one-time setup instructions. After that's been done at least once, run the following (if you're using Windows, you'll need to change the directory to `~\pack\plugins\start`) and restart Vim:
>
> ```
> $ git clone https://github.com/easymotion/vim-easymotion \
>   ~/.vim/pack/ plugins/start/vim-easymotion
> ```

After installing it, you can invoke the plugin by hitting the leader key (\) twice, followed by the desired movement key.

> **Follow the leader!**
>
> The leader key is often used by plugins to provide additional key mappings. By default, Vim's leader key is \. We'll go into more detail about the leader key in *Chapter 3*, *Follow the Leader – Plugin Management*.

Try using it with a word-wise motion by invoking *w* (backslash, followed by a backslash, followed by *w*):

Figure 2.33 – EasyMotion plugin allows you to instantly move to any text object on the screen.

You can see that the beginning of every word on the screen has been replaced with a letter (or two letters, once EasyMotion runs out of letters from the English alphabet). Pressing the letter (or two in order) will instantly transport your cursor to that spot on the screen.

In the example above, pressing *j* followed by *l* will take your cursor to the `has_spam` word within the `prepare_ingredient` method:

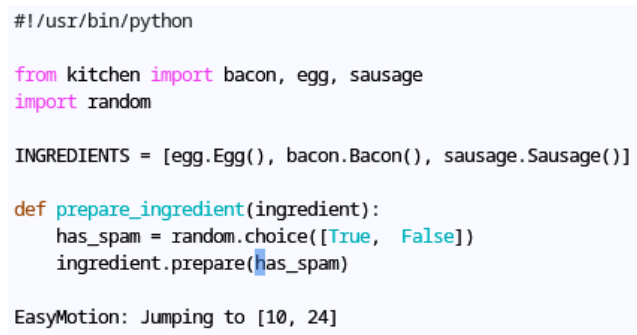

Figure 2.34 – Tada! Jumping to the exact position with EasyMotion is instant!

EasyMotion supports the following movement commands by default (all prefixed by double-tapping the leader key):

- *f*, to look for a character to the right and *F* to look for the character on the left
- *t*, to move until the character on the right and *T* until the character on the left
- *w*, to move by word (and *W* by WORD)
- *b*, to move backward by word (and *B* by WORD)
- *e*, to move forward to the end of the word (and *E* for the WORD)
- *ge*, to move backward to the end of the word (and *gE* for the WORD)
- *k* and *j* to go to the beginning of the line up or down
- *n* and *N* for jumping through search results on the page (based on the last / or *?* search)

EasyMotion leaves many keys unassigned, leaving it up to the user to build their own set of mappings. You should check `:help easymotion` to see everything EasyMotion can do.

Copying and pasting with registers

You can copy text by using the *y* (**y**ank) command, followed by a movement or a text object. You can also hit *y* from a visual mode when you have selected some text.

> **Tip**
> In addition to all of the standard movement, you can use *yy* to **y**ank the contents of the current line.

Let's yank the following piece of code by typing *ye* (**y**ank until the end of the **w**ord):

```
#!/usr/bin/python

from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    ingredient.prepare()
```

Figure 2.35 – *ye* will **y**ank (copy) `has_spam` into the default register.

This will copy `has_spam` into our default register. Now, place the cursor where you want the text to appear (the text is inserted after the cursor):

```
#!/usr/bin/python

from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    ingredient.prepare()
```

Figure 2.36 – Place cursor before the position where you want to insert the text.

To **p**aste the code, hit *p*:

```
#!/usr/bin/python

from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    ingredient.prepare(has_spam)
```

Figure 2.37 – *p* will insert the default register contents after the cursor.

The delete and change operators also yank content so that you can paste it later. Oh, and you can prefix the paste command with a number, in case you ever want to duplicate something multiple times.

Where do the registers come in?

Whenever you copy and paste text, it's saved in a register. Vim allows you to operate with many registers, which are identified by letters, numbers, and special symbols.

Registers can be accessed by hitting ", followed by the register identifier, followed by the operation on said register.

Registers `a-z` are used for holding manually assigned data. For example, to yank a word into the `a` register, you can run *"ayw* and paste it using *"ap*.

Macros

Registers are also used to record macros, which you will learn about in *Chapter 6*, *Refactoring Code with Regular Expressions and Macros*.

All of the operations you've performed so far have used the unnamed register. If you ever need to access the unnamed register explicitly, it is identified by a double quote character, `"`. For example, you can use *""p* to paste from the unnamed register (which is the same as just invoking *p*).

Numbered registers allow you to access the contents of your yank and delete operations. `0` is the last yanked text, while registers `1–9` access the last deleted text. For example, if you have a stellar memory, you can paste some text you deleted seven operations ago by hitting *"7p*.

Read-only registers

There are some read-only registers you might find handy: `%` holds the name of the current file, # holds the name of the previously opened file, `.` is the last inserted text, and `:` is the last executed command.

You can also interact with registers from outside of a normal mode. *Ctrl + r* is a convenient shortcut, which allows you to paste a register's contents when you're in insert or command-line modes. For example, while you're in insert mode, *Ctrl + r*, " will paste the contents of the unnamed buffer at the position of the cursor.

You can access the content of a register at any time by running `:reg <register names>`. For instance, if you wanted to check what's inside the `a` and `b` registers, you'd run `:reg a b`. Here's the output:

```
INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    ingredient.prepare(has_spam)

:reg ab
Type Name Content
  c  "a   has_spam
  c  "b   prepare_ingredient(ingredient):
Press ENTER or type command to continue
```

Figure 2.38 – `:reg a b` lists contents of registers `a` and `b`. This is very handy when you're copying and pasting multiple things and want to keep your head straight.

In the preceding example, the `a` register contains `has_spam`, and the `b` register contains `prepare_ingredient(ingredient)`.

In addition, you can list the contents of every register by running `:reg` without any parameters.

Named registers (`a–z`) can be appended to as well. To append to a register instead of overwriting it, capitalize the register name. For example, to append a **word** to the register `a`, run *"aye* with a cursor at the beginning of the word.

Copying from outside of Vim

There are two built-in registers that interact with the outside world:

- The `*` register is the primary system clipboard (the default clipboard in Mac and Windows, and mouse selection inside a Terminal in Linux)
- The + register (only in Linux) is used for Windows-style *Ctrl* + *c* and *Ctrl* + *v* operations (referred to as **Clipboard selection**)

> **This doesn't always work**
>
> For the clipboard to interact with the outside world, Vim must be compiled with the `+clipboard` option (and on Unix with `+xterm_clipboard` support as well). You can check if Vim is compiled with the `+clipboard` option by running `vim --version | grep clipboard` in the command line. See the *Compilation options* section in *Chapter 1*, *Getting Started*, to learn more about compiling Vim with different options.
>
> It also often doesn't work when SSHing, requiring terminal emulator-specific configuration to function.

You can interact with these registers as you would with any other. For instance, you can use *"*p* to paste from the primary clipboard or use *"+yy* to yank a line into Linux's Clipboard selection.

If you want Vim to work with these registers by default, you can set the `clipboard` variable in your `.vimrc` file. Set it to `unnamed` to copy and paste from the `*` register:

```
set clipboard=unnamed " Copy into system (*) register.
```

Set it to `unnamedplus` for yank and paste commands to work with the + register by default:

```
set clipboard=unnamedplus " Copy into system (+) register.
```

You can also tell Vim to use both at once:

```
set clipboard=unnamed,unnamedplus " Copy into system (*, +) register.
```

Now, *y* and *p* will yank and paste from the specified register by default.

Pasting in insert mode

You can also sometimes choose to paste text from the system clipboard while in the insert mode. In older Vim versions or in certain Terminal emulators, this will yield some issues, since Vim will try to automatically indent code or extend commented-out sections. To avoid this, run `:set paste` before pasting code to disable auto indent and auto comment insertion. Run `:set nopaste` to turn it back on once you're done.

Most of these issues are resolved in bracketed paste mode, which is enabled by default, starting with version 8.0. See `:help xterm-bracketed-paste` for more details.

Summary

You now know how to navigate core concepts Vim operates by using buffers to represent files, utilizing split windows, and using tabs to organize multiple windows. You've also learned how to use folds to make navigating large files more manageable.

You now should be more confident getting through a large code base by navigating files with plugins such as Netrw, NERDTree, Vinegar, and CtrlP. Oh, and this chapter taught you a quick (even though it's a slightly manual) way to install said plugins.

This chapter covered new movement operations, text objects, ways to quickly dart into insert mode, and how to make even fancier jumps throughout the file using the EasyMotion plugin. We've also dipped into search functionality, searching both within a single file and across the whole code base. You get a bonus point for trying out the `ack` plugin.

Finally, this chapter covered the concept of registers, and how you can use them to supercharge copying and pasting text.

In the next chapter, we'll take a deeper look at plugin management, and we'll go into detail about modes in Vim, as well as creating custom mappings and commands.

3

Follow the Leader Plugin Management

Vim plugins are easy to make, and the number of available plugins keeps growing every year. Some cater to a very narrow audience and improve on a very particular workflow, while others aim to make Vim more effective to use for the public. This chapter will take a deep dive into installing plugins and customizing your workflow through remapping keys. This chapter will cover the following topics:

- Ways to manage multiple plugins with vim-plug, Vundle, Pathogen, or Vim's native package feature
- A way to profile slow plugins
- An in-depth explanation of primary modes in Vim
- Intricacies of remapping commands
- The leader key and how it's useful for all kinds of custom shortcuts
- Configuring and customizing plugins

Technical requirements

In this chapter, you will be working on your `.vimrc` file. If you get lost through this chapter, you can get the resulting file from GitHub: `https://github.com/PacktPublishing/Mastering-Vim-Second-Edition/tree/main/Chapter03`.

Managing plugins

You have already installed quite a few plugins so far, and the number will only keep going up, especially as you try to solve problems that are specific to whatever it is you're working on. Manually keeping plugins up to date requires quite a bit of work, but luckily there are plugin management solutions out there!

Plugin management becomes even more important if you often change or switch machines and have the need to keep multiple plugins updated.

> **Cross-platform environments**
>
> For more tips on keeping Vim synced between multiple machines, see *Chapter 7*, *Making Vim Your Own*.

The plugin management landscape is ever changing, and there's no substitute for good old research when choosing a plugin manager. This chapter covers a few plugin managers that I've used throughout the years, which will hopefully be enough for you to base your own research on. Let's dig in.

vim-plug

The newest and the brightest in plugin management is **vim-plug**, a lightweight plugin that makes it easy to deal with a multitude of plugins. The plugin is available on GitHub at `https://github.com/junegunn/vim-plug` (it has a rather friendly `README` file, but I've captured the gist of it in this section if you're feeling lazy).

There are some neat things about this plugin:

- It's lightweight and fits in a single file, allowing for some straightforward installation options
- It supports parallel plugin load (if Vim is compiled with Python or Ruby enabled, which is true for all modern Vim setups)
- It supports the lazy loading of most plugins, only invoking plugins for a particular command or a file type

> **You'll want to start fresh!**
>
> In the previous chapter, you manually installed the plugins. This section provides a much better plugin experience, and you'll want to delete the plugins you downloaded manually. In order to do that, remove the manually added plugin directory (`rm -rf $HOME/.vim/pack` in Linux and `rmdir /s %USERPROFILE%\vimfiles\pack` in Windows).

Installing vim-plug is straightforward:

1. Fetch the plugin file from `https://raw.github.com/junegunn/vim-plug/master/plug.vim`.
2. Save the file as `$HOME/.vim/autoload/plug.vim`.

> **Fetching a file from GitHub**
>
> To fetch a single file from GitHub, you can use `curl` or `wget` on Linux or macOS, or just open the link in the browser, right-click, and choose **Save as...**. For instance, you could run the following command to fetch the file in Unix:
>
> ```
> $ curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
>   https://raw. github.com/junegunn/vim-plug/master/plug.vim
> ```

3. Update your `.vimrc` file to include vim-plug initializers:

```
" Manage plugins with vim-plug.
call plug#begin()
call plug#end()
```

4. Add some plugins between these two lines, using the last parts of the URL in GitHub (in `<username>/<repository>` format, for example, `scrooloose/nerdtree` instead of `https://github.com/scrooloose/nerdtree`) to identify the plugins:

```
" Manage plugins with vim-plug.
call plug#begin()

Plug 'scrooloose/nerdtree'
Plug 'tpope/vim-vinegar'
Plug 'ctrlpvim/ctrlp.vim'
Plug 'mileszs/ack.vim'
Plug 'easymotion/vim-easymotion'

call plug#end()
```

5. Save your `.vimrc` file and reload it (`:w | source $MYVIMRC`) or restart Vim to apply the changes. Execute `:PlugInstall` to install the plugins.

This will download the aforementioned plugins from GitHub:

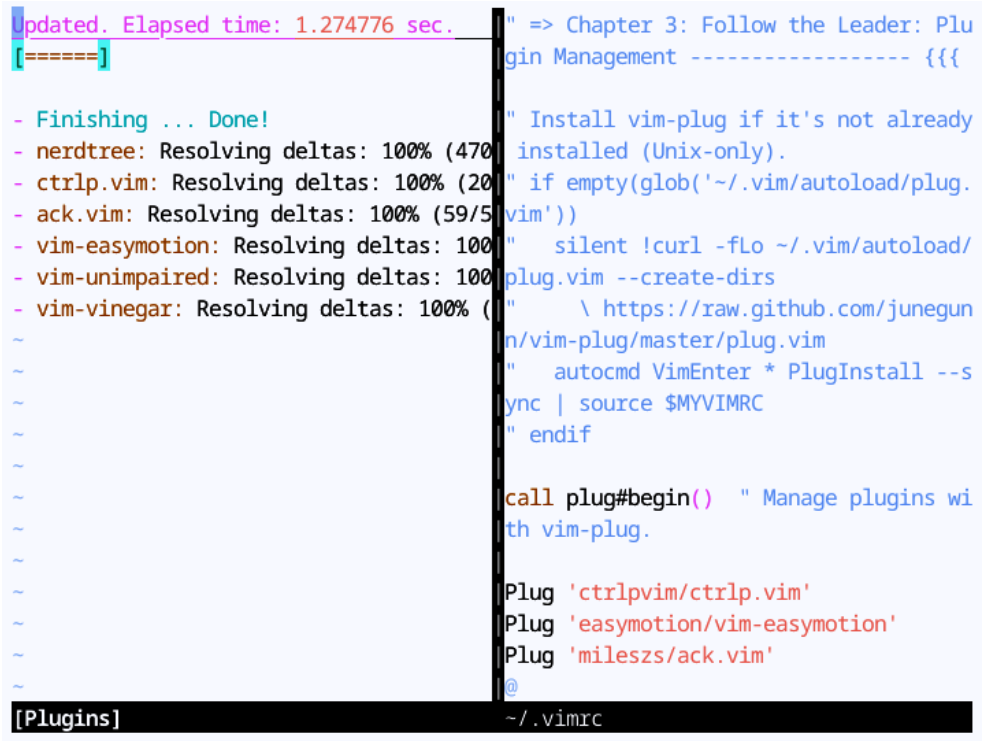

Figure 3.1 – Installing plugins with vim-plug

There are two main commands you will use with vim-plug:

- `:PlugUpdate` will update all of the plugins you have installed.
- `:PlugClean` will delete the plugins you removed from your `.vimrc` file. If you don't run `:PlugClean`, the plugins you deactivated (by either commenting out or removing the relevant `Plug ...` line in your `.vimrc` file) will stay on your system.

> **On updating vim-plug itself**
>
> Running `:PlugUpdate` updates every plugin except for vim-plug itself. If you want to update vim-plug, you need to run `:PlugUpgrade` and reload your `.vimrc` file by either running `:source $MYVIMRC` or restarting Vim.

Lazy plugin loading is a useful tool to prevent plugins from slowing down Vim. The `Plug` directive supports optional parameters. For instance, if you wanted to load NERDTree when the `:NERDTreeToggle` command is called, you could use the `on` parameter:

```
Plug 'scrooloose/nerdtree', { 'on': 'NERDTreeToggle' }
```

Lazy loading might not always work

Depending on the plugin, lazy loading may or may not work well. If you're running into an issue with a plugin, consider first disabling lazy loading.

If you wanted to only load a plugin for a particular file type, you could use the `for` parameter:

```
Plug 'junegunn/goyo.vim', { 'for': 'markdown' }
```

Getting vim-plug :help to work

Due to the way vim-plug is installed, its help pages are not available by default. If you'd like to be able to call `:help vim-plug`, add `Plug 'junegunn/vim-plug'` to a list of installed plugins and run `:PlugInstall`.

You can find the list of supported parameters in vim-plug's `README` file on GitHub at `https://github.com/junegunn/vim-plug`.

You can add the following piece to your `.vimrc` file to install vim-plug whenever you transport your `.vimrc` file to a new machine:

```
" Download and install vim-plug (Linux, Mac, and Windows).
if empty(glob(
    \ '$HOME/' . (has('win32') ? 'vimfiles' : '.vim') .
      \ '/autoload/plug.vim'))
  execute '!curl -fLo ' .
    \ (has('win32') ? '\%USERPROFILE\%/vimfiles' : '$HOME/.vim') .
    \ '/autoload/plug.vim --create-dirs ' .
    \ 'https://raw.githubusercontent.com/junegunn/vim-plug/master/
plug.vim'
  autocmd VimEnter * PlugInstall --sync | source $MYVIMRC
endif
```

This will install vim-plug (and all the plugins you have listed) the next time you open Vim.

Alternatives to vim-plug

There are many more plugin management alternatives to vim-plug. The following list is not in any way comprehensive, but it highlights different styles of plugin management. Pick the one that's more suited to your taste, or maybe search the web for alternatives.

Do it yourself – Vim packages

You could always decide to take the do-it-yourself route and implement your own solution for storing plugins. That's what we effectively did in the previous chapter, albeit with fewer bells and whistles.

Since most plugins are available on GitHub, a popular way of making sure that the plugins are up to date is installing them as Git submodules. If you're familiar with Git, you can initialize a repository in your `.vim` folder and install plugins as submodules.

Vim 8 introduced a native way to load plugins, by expecting the files to be in a directory tree under `.vim/pack`. Vim 8 expects the following structure of the files:

- `.vim/pack/<any-directory-name>/opt/` is used for plugins you want to manually load
- `.vim/pack/<any-directory-name>/start/` is used for plugins you always load

> **The nitty gritty details**
> If you're the curious type, you could learn more about how each individual plugin folder is structured in *Chapter 8*, *Transcending the Mundane with VimScript*.

You may want to use some explicit name for a directory under `.vim/pack/`. For instance, `plugins` might be a good choice.

You can use the `start/` directory for plugins that you always want to load.

On the other hand, `opt/` only loads plugins when you execute `:packadd <plugin-directory-name>`. This lets you add `packadd` commands to your `.vimrc` file. Using the `opt/` and `packadd` commands lets you achieve plugin lazy-loading (just like vim-plug):

```
" Load and run ack.vim plugin on :Ack command.
command! -nargs=* Ack :packadd ack.vim | Ack <f-args>
" Load an run Goyo plugin when opening Markdown files.
autocmd! filetype markdown packadd goyo.vim | Goyo
```

> **On version control**
> If you decide to choose this route, do visit *Chapter 7*, *Making Vim Your Own*, which will cover some best practices when it comes to version controlling your Vim configuration.

In addition, you'll want to add the following two lines to your `.vimrc` file to load the documentation for all the plugins:

```
packloadall " Load all plugins.
silent! helptags ALL " Load help for all plugins.
```

`packloadall` tells Vim to load every plugin in the `start/` directory (Vim automatically performs this step after `.vimrc` is loaded, but we want to call it earlier). `helptags ALL` loads all available help entries for our plugins, and the `silent!` prefix hides any output and errors you might receive when loading the help entries.

You can manage your plugins yourself (with some overhead) by using Git submodules to download the plugins and keep them up to date.

Initialize a Git repository in the `.vim` directory (a one-time step) with the following command:

```
$ cd ~/.vim
$ git init
```

Add a plugin as a submodule:

```
$ git submodule add \
  https://github.com/scrooloose/nerdtree.git \
  pack/plugins/start/nerdtree
$ git commit -am "Add NERDTree plugin"
```

Now, every time you want to update your plugins, you can run the following:

```
$ git submodule update --recursive
$ git commit -am "Update plugins"
```

To delete a plugin, remove the submodule with the following steps:

```
$ git submodule deinit -f -- pack/plugins/start/nerdtree
$ rm -rf .git/modules/pack/plugins/start/nerdtree
$ git rm -f pack/plugins/start/nerdtree
```

Do it yourself is a great route to take if you're a tinkerer, a minimalist, or otherwise enjoy making your life harder than it needs to be.

Vundle

Vundle is vim-plug's predecessor (and possibly an inspiration), which works along similar lines. Plugin installation is synchronous, and the plugin packs slightly more weight than vim-plug. Vundle and its installation instructions are available from GitHub at: `https://github.com/VundleVim/Vundle.vim`.

Vundle works similar to vim-plug, with the `:PluginInstall`, `:PluginUpdate`, and `:PluginClean` commands performing the same functions.

Vundle's differentiating feature used to be the `:PluginSearch` command, which enabled you to search and try out plugins from within Vim, but that feature stopped working back in 2019 and hasn't been fixed.

Pathogen

By definition, **Pathogen** is a `runtimepath` manager and not a plugin manager. However, in practice, `runtimepath` manipulation translates into plugin management really well. After Vim 8.0, there's no need to manipulate `runtimepath` to install plugins. However, if you have to use Vim prior to 8.0 (and you don't want to use full-blown package managers), Pathogen might make your life noticeably easier.

Pathogen was one of the earlier takes on plugin management and has heavily influenced the landscape of its successors. Many Vim users still use it today, but the influx of new adopters has stopped.

Pathogen is available from GitHub at `https://github.com/tpope/vim-pathogen`.

Profiling slow plugins

As you use Vim a lot, you might end up with numerous plugins installed. Sometimes, these plugins can cause Vim to become slow. Often, the culprit is a single unoptimized plugin, either due to the author's oversight or the unique way the plugin interacts with your system. Vim comes with built-in profiling support, which we'll now learn to use.

Profiling startup

You can start Vim with a `--startuptime <filename>` flag, which will log every action Vim takes during startup into a file. For instance, here's how you write the startup log into `startuptime.log`:

```
$ vim --startuptime startuptime.log
```

> **For gvim users**
>
> You can launch gvim in a similar manner with `gvim --startuptime startuptime.log`. The commands are the same in the Linux command line and in Windows `cmd.exe`.

Quit Vim and open `startuptime.log`. You'll be greeted with something like this (I replaced sections of the file with `<...>` to make it easier to read):

```
times in msec
 clock   self+sourced   self:  sourced script
 clock   elapsed:              other lines

000.021  000.021: --- VIM STARTING ---
000.244  000.223: Allocated generic buffers
<...>
008.902  003.923  003.923: sourcing /usr/local/share/vim/vim90/colors/lists/de
009.526  005.150  001.227: sourcing /usr/local/share/vim/vim90/syntax/syncolor
<...>
040.614  036.899  003.242: sourcing $HOME/.vimrc
040.617  000.528: sourcing vimrc file(s)
041.195  000.196  000.196: sourcing /home/ruslano/.vim/plugged/ctrlp.vim/autol
041.413  000.621  000.425: sourcing /home/ruslano/.vim/plugged/ctrlp.vim/plugi
047.011  005.512  005.512: sourcing /home/ruslano/.vim/plugged/vim-easymotion/
047.299  000.207  000.207: sourcing /home/ruslano/.vim/plugged/ack.vim/plugin/
056.223  008.857  008.857: sourcing /home/ruslano/.vim/plugged/vim-unimpaired/
056.613  000.315  000.315: sourcing /home/ruslano/.vim/plugged/vim-vinegar/plu
056.975  000.073  000.073: sourcing /usr/local/share/vim/vim90/plugin/getscrip
057.194  000.200  000.200: sourcing /usr/local/share/vim/vim90/plugin/gzip.vim
057.441  000.228  000.228: sourcing /usr/local/share/vim/vim90/plugin/logiPat.
057.499  000.038  000.038: sourcing /usr/local/share/vim/vim90/plugin/manpager
057.716  000.199  000.199: sourcing /usr/local/share/vim/vim90/plugin/matchpar
```

Figure 3.2 – Output of running vim with a `--startuptime` flag

In the preceding screenshot, you can see a set of timestamps (most in three columns), followed by an action measured by the timestamp. The timestamps are in milliseconds: 1/1000 of a second. The first column indicates the number of milliseconds from starting Vim, while the last column indicates how long each action took.

It's the last column that is of interest to us. You'll be looking for any abnormalities in the file.

In our case, we don't have any particularly slow plugins, but for the sake of science, the slowest plugin we have installed is **vim-unimpaired** (at 8 milliseconds—008.857). You'll have to get closer to 500 milliseconds (or half a second) for the plugin to have a noticeable effect on Vim startup times.

Profiling specific actions

If performing a particular action in Vim is slow, you can profile just a particular set of actions.

In this example, I created an obvious performance culprit. I downloaded the Python repository from GitHub (by running `git clone https://github.com/python/cpython.git`), and tried running the `:CtrlP` command that's provided by a CtrlP plugin from within the `cpython/` directory (which we explored in *Chapter 2*, *Advanced Editing and Navigation*). `:CtrlP` will try to index all the files recursively starting at the current directory, which should be slow for such a large number of files.

Start Vim as usual, and execute the following set of commands:

```
:profile start profile.log
:profile func *
:profile file *
```

From now on, Vim will profile every action you perform. Run the slow command. In our case, let's run `:CtrlP` by pressing *Ctrl* + *p*. After the offending action has been performed, quit Vim (`:q`).

Open `profile.log`, and you'll be greeted with something like this (you may want to have folds enabled with `:set foldmethod=indent`, as the file is large and hard to navigate otherwise):

```
SCRIPT  /home/ruslano/.vim/plugged/ctrlp.vim/autoload/ctrlp.vim
Sourced 1 time
Total time:   0.002072681
 Self time:   0.002048528

count     total (s)      self (s)
+--2900 lines: " ============================================================

SCRIPT  /home/ruslano/.vim/plugged/ctrlp.vim/autoload/ctrlp/utils.vim
Sourced 1 time
Total time:   0.000162402
 Self time:   0.000089125

count     total (s)      self (s)
+--119 lines: " ============================================================

FUNCTION  <SNR>3_SynSet()
    Defined: /usr/local/share/vim/vim90/syntax/synload.vim:33
Called 1 time
Total time:   0.000152604
 Self time:   0.000152604

count     total (s)      self (s)
+-- 28 lines: " clear syntax for :set syntax=OFF  and any syntax name that doesn
```

Figure 3.3 – `profile.log` contains a very detailed execution log

Scroll to the bottom of the file (by pressing *G*), and you'll see a list of functions sorted by how long they took to execute:

```
FUNCTIONS SORTED ON SELF TIME
count     total (s)      self (s)  function
50468                0.247723278  ctrlp#complen()
 5327                0.203391060  <SNR>32_usrign()
    8   1.584669354   0.191512642  <SNR>32_GlobPath()
    8   0.372570320   0.169122409  ctrlp#dirnfile()
    2   0.127770403   0.127694947  ctrlp#utils#writecache()
    1   0.918644511   0.103977994  ctrlp#files()
    1   0.005569569   0.005561682  ctrlp#rmbasedir()
  345                0.003468485  ctrlp#utils#fnesc()
  344   0.005172569   0.001711454  <SNR>32_fnesc()
   17   0.004104682   0.001254823  <SNR>32_mixedsort()
    1                0.001084234  <SNR>32_MapNorms()
    1   0.001260093   0.000905071  <SNR>32_Open()
    1                0.000716905  <SNR>32_MapSpecs()
    7   0.000683036   0.000659418  ctrlp#progress()
   34                0.000538268  <SNR>32_getparent()
    1   0.000734820   0.000519823  <SNR>32_opts()
    1                0.000480215  <SNR>32_sublist()
   17   0.001195986   0.000470162  <SNR>32_comparent()
   35                0.000421236  <SNR>32_CurTypeName()
    5   0.000410148   0.000367721  <SNR>24_Highlight_Matching_Pair()
```

Figure 3.4 – The bottom of the `profile.log` contains functions sorted by execution length

Many of the slowest functions are prefixed with `ctrlp#`, so it's becoming clear that CtrlP is likely to be the culprit of the slowness (and as we know—it is). If it is not explicitly obvious where the functions come from, we can search the file for the function name. For instance, `<SNR>32_GlobPath()` is responsible for nearly ~1.58 seconds of slowdown (hover over the function name and press * to search for the word under the cursor):

```
FUNCTION  <SNR>32_GlobPath()
    Defined: ~/.vim/plugged/ctrlp.vim/autoload/ctrlp.vim:453
Called 8 times
Total time:   1.584669354
 Self time:   0.191512642

count     total (s)      self (s)
    8                0.059575180       let entries = split(globpath(a:dirs, s:g
lob), "\n")
    8   0.372688941   0.000118621       let [dnf, depth] = [ctrlp#dirnfile(entri
es), a:depth + 1]
    8                0.000418963       let g:ctrlp_allfiles += dnf[1]
    8   0.000112344   0.000091020       if !empty(dnf[0]) && !s:maxf(len(g:ctrlp
_allfiles)) && depth <= s:maxdepth
    7   0.000741084   0.000058048           sil! cal ctrlp#progress(len(g:ctrlp_
allfiles), 1)
    7   0.006641475   0.001531737           cal s:GlobPath(join(map(dnf[0], 's:f
nesc(v:val, "g", ",")'), ','), depth)
    8                0.000004235       en

FUNCTION  <SNR>32_stop_timer_if_exists()
    Defined: ~/.vim/plugged/ctrlp.vim/autoload/ctrlp.vim:506
Called 1 time
```

Figure 3.5 – Function `<SNR>32_GlobPath()` is called by the CtrlP plugin, as evident from the CtrlP references throughout

As you can see by the number of CtrlP references, this function is likely related to the offending plugin.

If you were to try to profile real issues with Vim, the contents of `profile.log` could tell you which plugins are a likely cause of the slowdowns you are experiencing.

Whether you use vim-plug, Vundle, Pathogen, or even native plugin management out of the box, Vim plugins expand and tailor Vim functionality to you. Now, you know how to profile Vim if (when) all these plugins start slowing down Vim!

Deeper dive into modes

You've already encountered a few different modes Vim operates in, and this section will cover these and the remaining modes in depth. As you have already learned, Vim uses modes to know how to respond to inputs: a key press in normal mode will produce different results from a key press in insert or command-line mode.

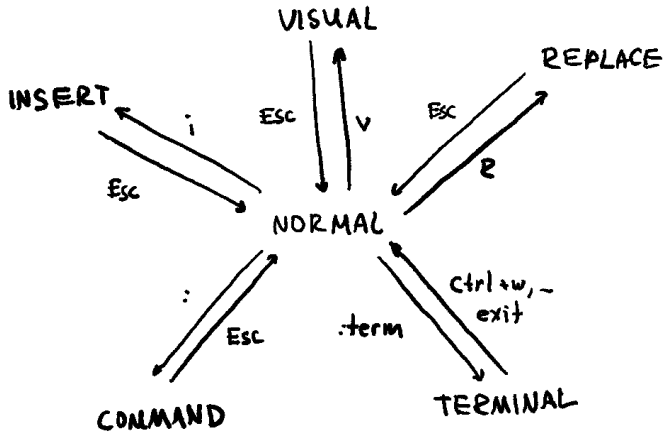

Figure 3.6 – Relationships between insert, visual, replace, command, and terminal modes

Vim has seven primary modes (you won't use them all), and it's important to understand what each mode does in order to comfortably navigate Vim.

Normal mode

This is where you will (and already did) spend most of your time with Vim. You enter **normal mode** by default when opening Vim, and you can go back to normal mode from other modes by pressing the *Esc* key (sometimes twice).

Command-line and ex modes

Command-line mode is entered by typing a colon (`:`) or when searching for text with `/` or `?`, and it allows you to input a command until you hit *Enter*. Command-line mode provides some useful shortcuts:

- The up and down arrows (or *Ctrl* + *p* and *Ctrl* + *n*) let you traverse command history one by one
- *Ctrl* + *b* and *Ctrl* + *e* let you go to the **b**eginning and the **e**nd of the line respectively
- The *Shift* or *Ctrl* keys combined with left or right arrows allow you to move by words

A highly useful shortcut is *Ctrl* + *f*, which opens an editable command-line window with a history of the commands you ran:

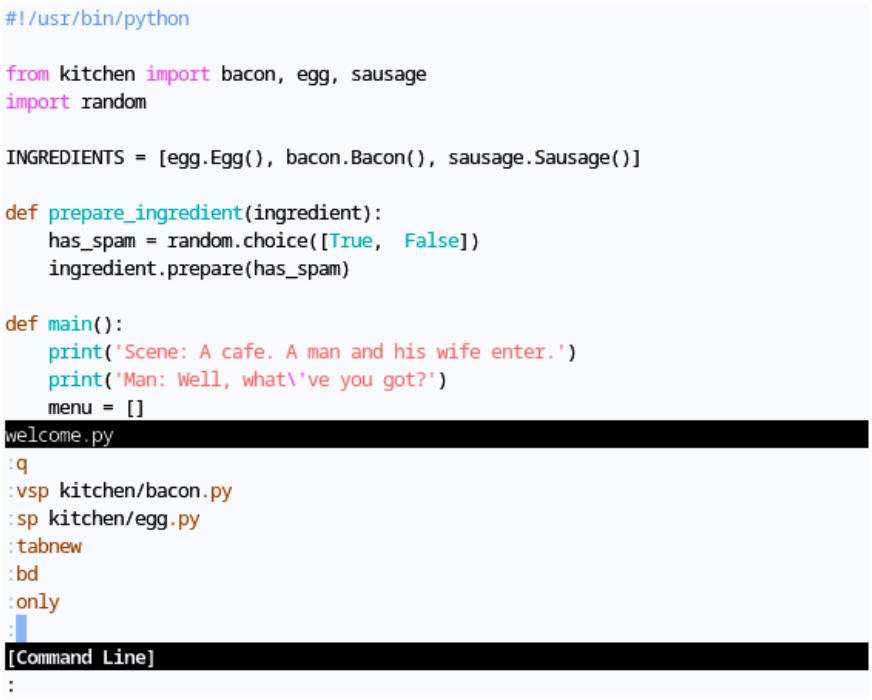

Figure 3.7 – Command-line mode history opened by hitting *Ctrl* + *f*

For editing purposes, it's a regular buffer, so you can find a command you've executed before, edit it (the way you'd edit any text in Vim), and execute it again. You can press *Enter* to execute the line your cursor is on or *Ctrl* + *c* to close the buffer.

You can learn more about working with command-line mode by looking up `:help cmdline-editing`.

Vim has a variation of the command-line mode called **ex mode**, which is entered by pressing *Q*. Ex mode is a compatibility mode with Vim's precursor—`ex`. It allows you to execute multiple commands without exiting the mode after each command, but it has very limited uses today.

Insert mode

Insert mode is used to type in text, and that's about it. Hitting *Esc* takes you back to normal mode, which is where you should be performing most of your work. When in insert mode, you can also use *Ctrl* + *o* to execute a single normal mode command and end up back in insert mode.

Insert mode is indicated by -- **INSERT** -- displayed in a status line.

Visual and select modes

Vim's **visual mode** allows for an arbitrary continuous selection of text (usually to perform some sort of operations on). It's useful when you want to work with a section of a file that does not map to existing text objects (word, sentence, paragraph, and so on). There are three ways to enter visual mode:

- *v* enters a character-wise visual mode (status line text: -- **VISUAL** --)
- *V* enters a line-wise visual mode (status line text: -- **VISUAL LINE** --)
- *Ctrl* + *v* enters a block-wise visual mode (status line text: -- **VISUAL BLOCK** --)

Once you enter visual mode, you are able to move your cursor using the usual movement commands, which would expand the selection. In the following example, we've entered a character-wise visual mode and moved the cursor by three words and one character to the right (by executing *3e* and then *l*). You can see `prepare_ingredient(ingredient)` being selected in visual mode:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
-- VISUAL --
```

Figure 3.8 – An example of selection in a visual mode

You can control the selection by doing the following:

- Pressing *o* to go to the other end of the highlighted text (hence allowing the selection to expand from the other side)
- Pressing *O* when in block-wise visual mode to the other end of the current line

After you're satisfied with the selection, you can run a command you'd like to execute on a selection. For example, hit *d* to delete the selected text:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def :
    has_spam = random.choice([True,  False])
    ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
```

Figure 3.9 – Deleting text selected in visual mode places you back in normal mode

In the preceding screenshot, Vim is back to normal mode (**-- VISUAL --** is not in a status line anymore), and the selection has been deleted. You can always hit *Esc* to come back to normal mode without making a change.

> **Text objects**
>
> Text objects become a powerful tool when used in visual mode. See *Utilizing Text Objects* in *Chapter 2*, *Advanced Editing and Navigation*, for more information.

Vim also has a **select mode**, which emulates the selection mode in other editors: pressing any printable character immediately erases the selected text and enters the insert mode (so the usual movement commands don't work here). Just like the ex mode, the select mode has a very specific and limited set of uses. In fact, I have never used it myself, except for when doing research for this book.

You can enter select mode by pressing *gh* from normal mode or *Ctrl* + *g* from visual mode and exit it by pressing (you guessed it) *Esc*.

Replace and virtual replace modes

Replace mode behaves similarly to those times when you accidentally press the *Insert* key on your keyboard and wonder why typing erases text. When working with replace mode, the text you type is placed over existing text (as opposed to moving the existing text in insert mode). This is great when you don't want to change the character count in the original line, for example.

Enter replace mode by hitting *R*:

```
    ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for ingredient in INGREDIENTS:
        menu.append(prepare_ingredient(ingredient))
    print('Waitress: Well, there\'s', ', '.join(menu))


-- REPLACE --
```

Figure 3.10 – Replace mode, as indicated by -- REPLACE -- in the status line

You'll see **-- REPLACE --** in the status line. Now, you'll be replacing text as you type:

```
    ingredient.prepare(has_spam)

def main():
    print('This is a spam skit bynd his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for ingredient in INGREDIENTS:
        menu.append(prepare_ingredient(ingredient))
    print('Waitress: Well, there\'s', ', '.join(menu))


-- REPLACE --
```

Figure 3.11 – Typing in replace mode replaces existing text (who would've thought?!)

Erasing in replace mode by hitting *Backspace* brings back the original text. Hit *Esc* to exit replace mode and go back to normal mode.

You can press *r* to enter replace mode for a single character press before being switched back to normal mode.

Vim also provides virtual replace mode, which behaves similarly to replace mode but operates in terms of screen real estate as opposed to characters in a file. The main noticeable differences include *Tab* replacing multiple characters (as opposed to a single character in replace mode) or *Enter* not creating an addition line, but moving on to the next line. You can enter virtual replace mode using `gR`, and you can learn more about it by reading `:help vreplace-mode`.

Terminal mode

Terminal mode came to Vim in version 8.1, and it allows you to run a terminal in a split window. You can enter terminal mode by typing the following:

```
:terminal
```

> **Tip**
> You can shorten the command to `:term`.

This will open your system's shell (your default shell in Linux or `cmd.exe` in Windows) in a horizontal split:

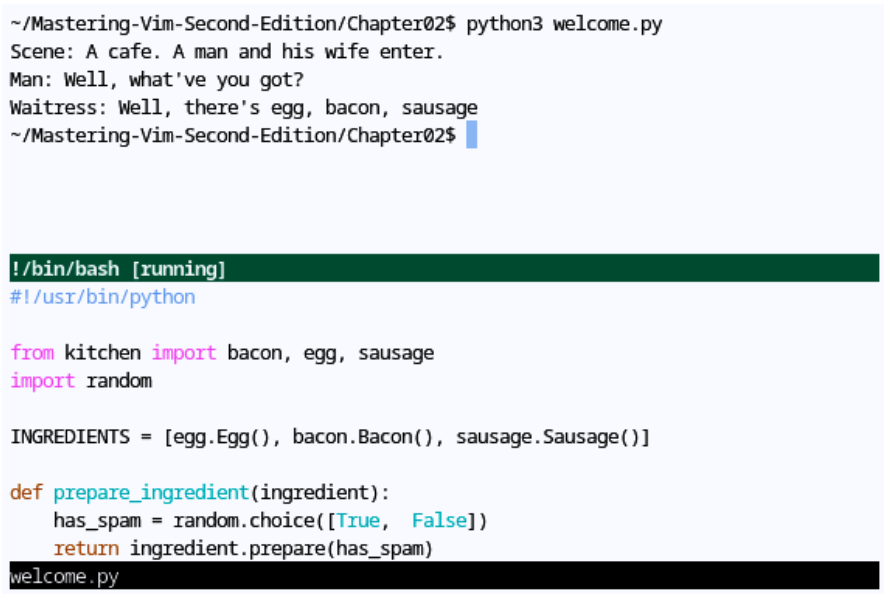

Figure 3.12 – `:terminal` enables you to work with a command-line terminal without exiting Vim (which we know to be notoriously difficult)

This is a wrapper around your system's terminal, which lets you work with your shell as you normally would. This window is treated similarly to any other window (you can navigate between split using *Ctrl* + *w* commands), but the window is effectively locked into insert mode. You may also want to consider using **tmux** or a **screen** under Linux or macOS to work with a Terminal alongside Vim.

You can also use `:term` to execute a single command and place its output in a buffer. For example, we can run `welcome.py` as follows:

```
:term python3 welcome.py
```

The output is immediately available to us in a horizontal split:

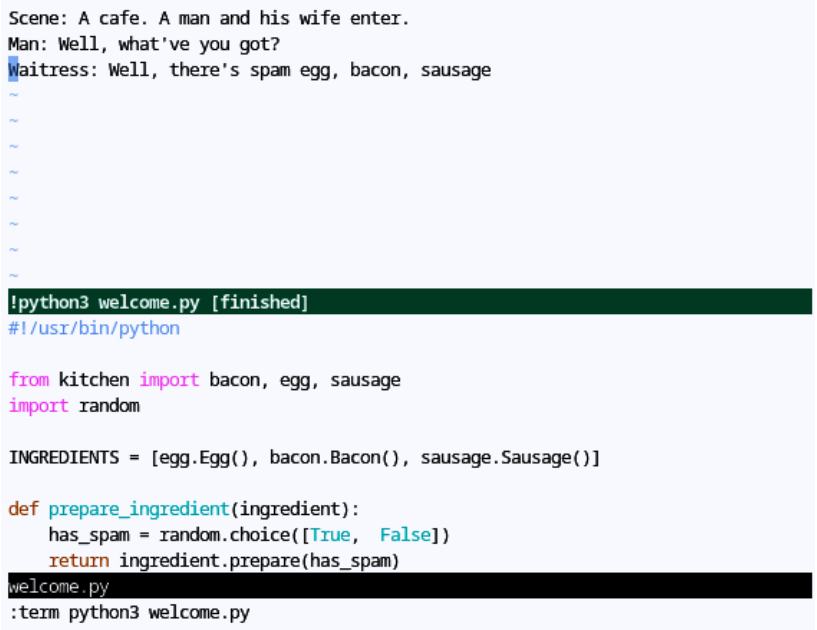

Figure 3.13 – Running `:term` with arguments executes a single command and captures output in a buffer

Operator-pending mode (bonus)

Finally, there's the operator pending mode. Every time you enter a command that expects a follow-up (e.g., when a command is followed by a motion, such as after hitting `y` when executing `yw` to yank a word), Vim enters an operator-pending mode.

You won't be deliberately using the operator-pending mode, but some commands only work in this mode, and it's helpful to be aware of the mode's existence.

Now, you're familiar with every major mode: normal mode, command-line and ex modes, insert mode, visual and select modes, terminal mode, and even operator pending mode. Thinking of which mode is needed to perform an editing task will be instrumental to your mastery of Vim!

Remapping commands

Now that you are comfortable working with plugins, you may want to consider customizing your Vim further by remapping commands to suit your preferences. Plugins are written by many kinds of different people, and everyone's workflow is different. Vim is infinitely extensible, allowing you to remap nearly every action, change certain default behaviors, and really make Vim your own. Let's talk about remapping commands.

Vim allows you to remap certain keys to be used in place of other keys. `:map` and `:noremap` provide just that:

- `:map` is used for recursive mapping in normal, visual/select, and operator-pending modes
- `:noremap` is used for non-recursive mapping

This means that commands remapped with `:map` are aware of other custom mappings, while `:noremap` works with system defaults.

> **Avoiding mapping collisions**
>
> Before you decide to create a new mapping, you may want to see whether the key or sequence you're mapping to is already used somewhere. You can scan through `:help index` for a list of built-in key bindings. The `:map` command lets you view plugin and user-defined mappings. For instance, `:map g` will display every mapping starting with the `g` key.

Let's add a custom mapping to our `.vimrc` file:

```
noremap ; : " Use ; in addition to : to type commands.
```

In the preceding example, we're remapping `;` to function the same way `:` does. Now, we don't have to press down *Shift* to enter command-line mode. On the downside, we now don't have a command that repeats the last `t`, `f`, `T`, or `F` (find character and find till character) movement.

We're using `noremap` because we still want to enter command-line mode, even if `:` gets remapped to do something else.

> **Cleaning the house**
>
> If you ever want to explicitly remove a mapping you or a plugin defined, you'd be looking for `:unmap`. There's also a nuclear option of `:mapclear`, which drops both user-defined and default mappings.

You can use special characters and commands in mappings as well, for example:

```
noremap <c-u> :w<cr> " Save using <Ctrl-u> (u stands for update).
```

`<c-u>` in the preceding example represents *Ctrl* + *u*. The *Ctrl* prefix in Vim is denoted by `<c-_>`, where `_` is some character. Other modifier keys are represented similarly:

- `<a-_>` or `<m-_>` represents *Alt* pressed with some key, for example, <m-b> would correspond to *Alt* + *b*
- `<s-_>` represents a *Shift* press, for example, <s-f> would correspond to *Shift* + *f*

Please note that a command is terminated by `<cr>`, which stands for a carriage return (the *Enter* key). Otherwise, the command will be entered but not executed, and you will be left hanging in command-line mode (unless that's exactly what you want).

By the way, here are all of the special characters you can use:

- `<space>`: *Spacebar*
- `<esc>`: *Esc*
- `<cr>`, `<enter>`: *Enter*
- `<tab>`: *Tab*
- `<bs>`: *Backspace*
- `<up>`, `<down>`, `<left>`, `<right>`: Arrow keys
- `<pageup>`, `<pagedown>`: *Page Up* and *Page Down*
- `<f1>` to `<f12>`: Function keys
- `<home>`, `<insert>`, `<del>`, `<end>`: *Home*, *Insert*, *Delete*, and *End*

:help!
See `:h key-notation` for the full list!

You can also map a key to `<nop>` (short for *no operation*) if you want the key to not do anything. This, for instance, could be useful if you're trying to get used to *hjkl*-style movement versus arrow keys. Disabling your arrow keys in `.vimrc` would look like this:

```
" Map arrow keys nothing so I can get used to hjkl-style movement.
map <up> <nop>
map <down> <nop>
map <left> <nop>
map <right> <nop>
```

Mode – aware remapping

The `:map` and `:noremap` commands work for normal, visual, select, and operator-pending modes. Vim supports a more fine-grained control over which modes the mappings work in:

- `:nmap` and `:nnoremap`: Normal mode
- `:vmap` and `:vnoremap`: Visual and select modes
- `:xmap` and `:xnoremap`: Visual mode
- `:smap` and `:snoremap`: Select mode

- `:omap` and `:onoremap`: Operator-pending mode
- `:map!` and `:noremap!`: Insert and command-line modes
- `:imap` and `:inoremap`: Insert mode
- `:cmap` and `:cnoremap`: Command-line mode

> **:help!**
> Vim often uses an exclamation mark `!` to force command execution or to add additional functionality to a command. Try `:help!` for yourself!

For example, if you wanted to add some mappings to alter some insert mode behavior, you could do this:

```
" Immediately add a closing quotes or braces in insert mode.
inoremap ' ''<esc>i
inoremap " ""<esc>i
inoremap ( ()<esc>i
inoremap { {}<esc>i
inoremap [ []<esc>i
```

In the preceding example, we changed the default behavior of a key press in insert mode (for example, the opening square bracket `[`) to insert two characters instead of one (`[]`), leave insert mode, and immediately reenter it (to be placed between both braces since insert mode is entered before the cursor).

The leader key

You've probably already encountered a key referred to as the **leader key**. The leader key is essentially a namespace for user- or plugin-defined shortcuts. Within a second of pressing the leader key, any key that's pressed will be in from that namespace.

The default leader key is a backslash, \, but it's not the most comfortable binding. There are a few alternative keys that are popular in the community, with the comma (,) being the most popular. To rebind the leader key, set the following in your `.vimrc` file:

```
" Map the leader key to a comma.
let mapleader = ','
```

You'll want to define your leader key closer to the top of `.vimrc` as the newly defined leader key will only apply to mappings defined after its definition.

> **Important note**
> When you rebind a key, its default functionality is overwritten. For example, comma (,) is used to replay the last *t*, *f*, *T*, or *F* movement commands, in the opposite direction.

My personal favorite is to use the spacebar as a leader key. It's a big key, which doesn't have any real use in normal mode (it mimics right arrow key functionality):

```
" Map the leader key to a spacebar.
let mapleader = "\<space>"
```

> **Important note**
>
> The escape character `\` is needed before space since `mapleader` doesn't expect special characters (such as `space`). Double quotes (") are also necessary for the escape to work since single quotes (') only allow literal strings.

You can use `leader` in your `.vimrc` mappings in the following manner:

```
" Save a file with leader-w.
noremap <leader>w :w<cr>
```

More often than not, you will use a leader key to map plugin functionality in a way that's easy for you to memorize, as in the following example:

```
noremap <leader>n :NERDTreeToggle<cr>
```

Configuring plugins

Plugins often expose commands for you to map to and variables to change plugin behavior. It's a good idea to review the available options and commands for the plugins you use. Sane plugin defaults make a huge difference in experience. Creating shortcuts that are easy for you to remember will help you remember how to use the plugin you forgot about in a few months.

Vim allows for the creation of global variables, which are primarily used to configure plugins. Global variables are usually prefixed by `g:`. You can find a list of configuration options in the plugin documentation by running `:help <plugin-name>` and looking for options.

For example, opening a help file for the CtrlP plugin (by running `:help ctrlp`) and searching for options (`/options`) yields the following:

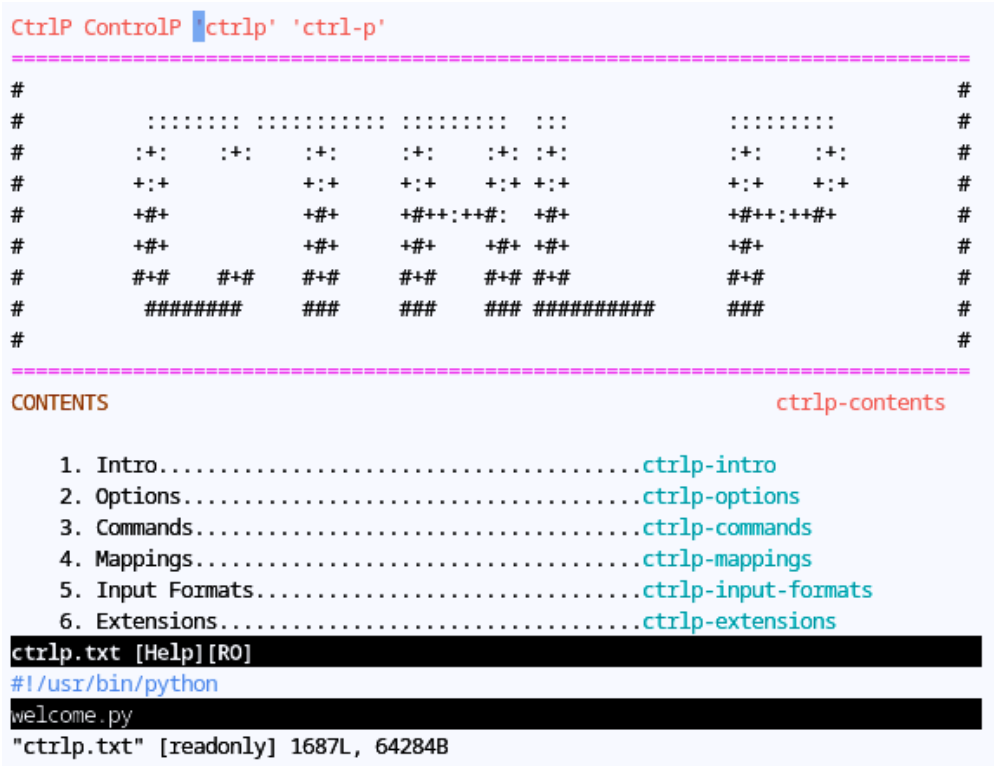

Figure 3.14 – The CtrlP help page

Refresher on CtrlP

You can read more about the CtrlP plugin under *Chapter 2*, *Advanced Editing and Navigation*.

If we follow the `ctrlp-options` link (by pressing *Ctrl* + *]*), we will be taken to a list of available options for configuring CtrlP:

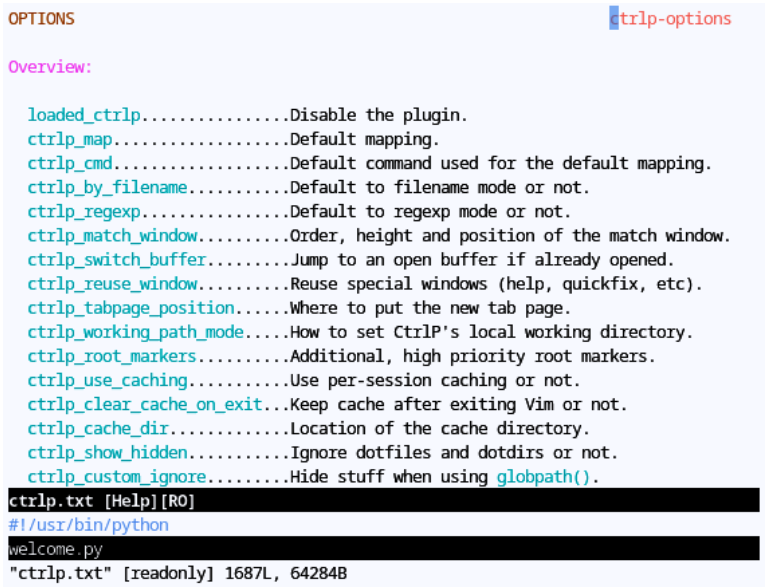

Figure 3.15 – Help page on the list of options available for configuring the CtrlP plugin

Let's pick an interesting option to explore, say `ctrlp_working_path_mode`. Move your cursor over to the link and press *Ctrl* + *]* to follow it further:

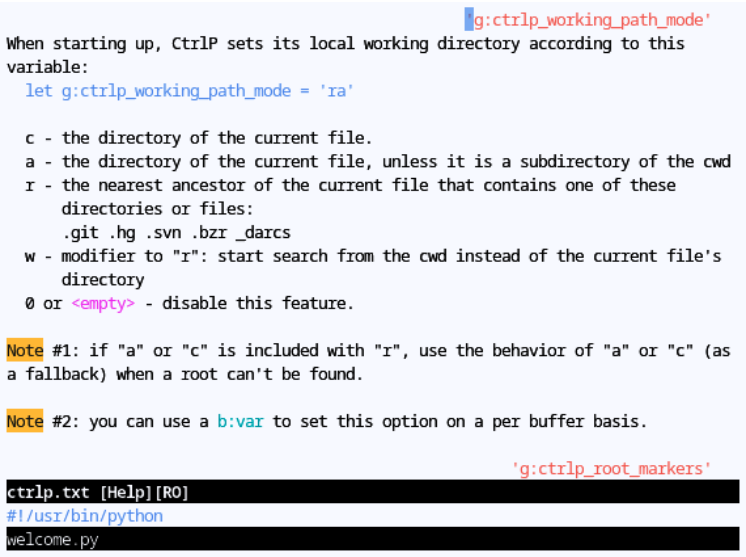

Figure 3.16 – A detailed help page on the `g:ctrlp_working_path_mode` option

It looks like this option guards how CtrlP sets a local working directory. If we wanted to change it to look for a root folder of our Git project (with a fallback to the current working directory), we would change our `.vimrc` file accordingly:

```
" Set CtrlP working directory to a repository root (with a
" fallback to current directory).
let g:ctrlp_working_path_mode = 'ra'
```

Digging through the available options for plugins takes time; however, it might make you much more productive or even completely change the way you use the plugins.

Remember the leader key? It comes in very handy with plugins as it provides a full namespace for plugins to use. Some plugins already use the leader key for their default key bindings, others not so much. You can always make your own easy-to-remember mappings!

For example, all three CtrlP modes can be easily accessed with two key presses:

```
" Remap CtrlP actions to be prefixed by a leader key.
noremap <leader>p :CtrlP<cr>
noremap <leader>b :CtrlPBuffer<cr>
noremap <leader>m :CtrlPMRU<cr>
```

I find it extremely useful to take some time to optimize my key mappings or customize plugin options. A small investment and some mindfulness go a long way in getting the most out of your setup.

Summary

In this chapter, we've talked about the different ways of managing plugins. The new shiny thing is vim-plug, a lightweight plugin manager that can asynchronously install and update your plugins. Vundle, its predecessor, also allows you to search for and temporarily install new plugins. We've also learned how to manually work with plugins: Vim 8.0 introduced a way to load plugins without the need to manually alter `runtimepath` for each plugin. If you still use Vim below version 8, then Pathogen provides a way to automate some of the `runtimepath` manipulations.

We've looked at profiling Vim with a `--startuptime` flag and the `:profile` command.

We've revisited modes, covering every major mode: normal mode, command-line and ex modes, insert mode, visual and select modes, and terminal mode.

We've talked about remapping commands to make Vim truly yours. Different key combinations are more convenient and easier to remember for different people. Vim allows you to remap keys based on the mode you're in, meaning you can alter the behavior of every key press. We've covered the leader key, which allows you to access a whole new namespace that's reserved for plugins and user-defined commands.

We've also looked into the ways we can squeeze the most out of our plugins by customizing their configuration options and adding key bindings that make more sense in our own unique workflow.

In the next chapter, we'll cover autocomplete, navigating large code bases with tags, and traversing Vim's undo tree.

4

Understanding Structured Text

At some point, text files grow large enough to be able to navigate through them top to down. Luckily, structured text such as code or Markdown files don't need to be read linearly. Vim has a few aces up its sleeve when it comes to navigating complex files. To assist with understanding your way around large bodies of text, this chapter will cover the following topics:

- Autocompleting code using Vim's built-in autocomplete functionality and plugins such as YouCompleteMe to enhance writing efficiency and accuracy
- Navigating large code bases using Exuberant Ctags, enabling instant movement between definitions and references
- Navigating Vim's complex undo tree with plugins such as Undotree, making it easier to navigate and recover changes

Technical requirements

We will continue navigating our sample project, and you will continue working on your `.vimrc` file. All of the material is available from GitHub at the following link: `https://github.com/PacktPublishing/Mastering-Vim-Second-Edition/tree/main/Chapter04`.

Code autocomplete

One of the most appealing features that modern IDEs have is code autocomplete. IDEs allow you to eliminate typos, look up hard-to-remember variable names, and save time by removing the need to type long variable names over and over again.

Vim has some built-in autocomplete functionality, and there are plugins that expand upon this.

Built-in autocomplete

Vim supports native autocomplete based on words available in open buffers. It's available out of the box starting with Vim 7.0. Start by typing the beginning of a function name and hit *Ctrl* + *n* to bring up the autocomplete list. You can navigate the list using *Ctrl* + *n* and *Ctrl* + *p*. For example, open `welcome.py`, enter insert mode, and start typing the first two letters of a function name: `pr` (`prepare_ingredient`). Press *Ctrl* + *n*. This will bring up a list of available options:

Figure 4.1 – Vim's native autocomplete

Continue typing to dismiss the list.

In fact, Vim has an insert-completion mode, which supports multiple completion types. While in insert mode, press *Ctrl* + *x* followed by one of the following keys:

- *Ctrl* + *l* to complete the whole line
- *Ctrl* + *]* to complete tags
- *Ctrl* + *f* to complete filenames
- *s* (with optional *Ctrl* this time around) to complete spelling suggestions (if `:set spell` is enabled)

There's more!

These are the commands I found useful in the past, but there are more! Read `:help ins-completion` for a full list of supported commands—everyone's workflow is unique, and you never know which commands you'll find yourself utilizing a lot. You should also check `:help 'complete'`, which is an option that controls where Vim looks for completion (by default, Vim looks in buffers, tag files, and headers).

YouCompleteMe

YouCompleteMe takes a built-in autocomplete engine and pumps steroids into it. YouCompleteMe has a few distinctive features that elevate it beyond built-in autocomplete:

- Semantic (language-aware) autocomplete; YouCompleteMe understands your code a lot better than built-in autocomplete
- Intelligent suggestion ranking and filtering
- An ability to display documentation, rename variables, autoformat code, and fix certain types of errors (language dependent, see `https://github.com/ycm-core/YouCompleteMe#quick-feature-summary`)

Installation

First, make sure that you have `cmake` and `llvm` installed, since YouCompleteMe needs to be compiled:

```
$ sudo apt install cmake llvm
```

For Windows, you can get `cmake` from `https://cmake.org/download` and `llvm` from `https://releases.llvm.org/download.html`.

YouCompleteMe relies on Python

YouCompleteMe requires Vim to be compiled with `+python3`. You can check whether your Vim was compiled with Python support by running `vim --version | grep python3`. If you see `-python3`, you'll have to recompile your Vim with Python support.

If you're using `vim-plug`, add the following to `.vimrc` between the `call plug#begin()` and `call plug#end()` lines:

```
let g:plug_timeout = 300 " Increase vim-plug timeout for
                         " YouCompleteMe.
Plug 'ycm-core/YouCompleteMe', { 'do': './install.py' }
```

Save the file and execute it:

```
:source $MYVIMRC | PlugInstall
```

Depending on how fast your machine is, this might take a while. You will be greeted by a successful installation screen:

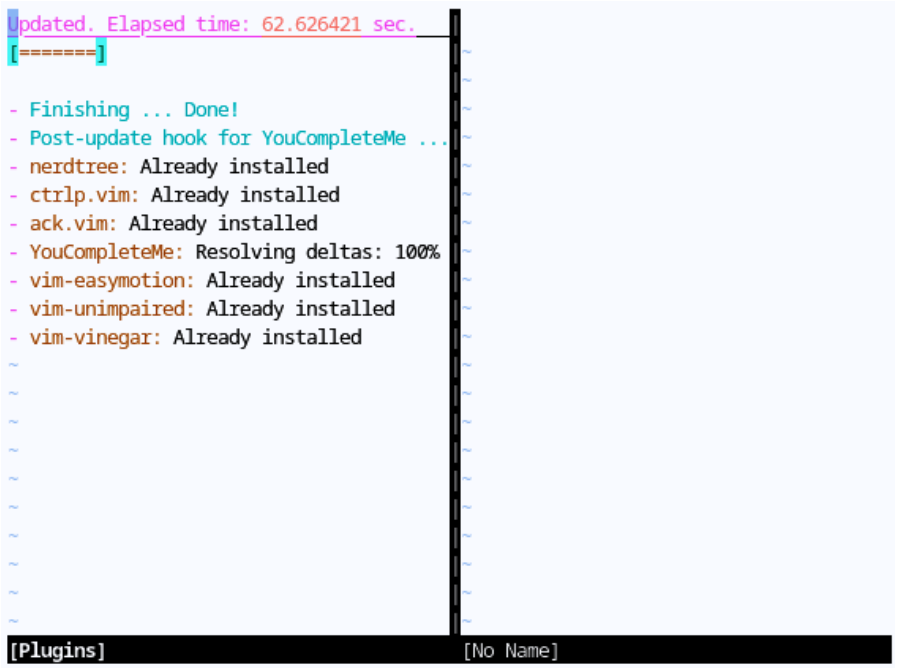

Figure 4.2 – Successfully installed YouCompleteMe

If you're running out of memory...

If you're running into an error such as **c++: internal compiler error: Killed (program cc1plus)**, then your machine likely doesn't have enough memory to complete the operation. On Linux, you can increase available swap space:

```
$ sudo dd if=/dev/zero of=/var/swap.img bs=1024k count=1000
$ sudo mkswap /var/swap.img
$ sudo swapon /var/swap.img
```

A swap file is a space on a hard drive that can be used to substitute the physical memory. Here, we create an empty 1 Gb swap file with the `dd` command, make the system recognize it as a swap file with `mkswap`, and activate the swap file with the `swapon` command.

Using YouCompleteMe

YouCompleteMe doesn't introduce a lot of new key bindings, which makes integrating it into your workflow easier. Enter insert mode and start typing away:

Figure 4.3 – YouCompleteMe shows autocomplete suggestions as you type

As you do, autocomplete suggestions will pop up. The *Tab* key will cycle through suggestions. Furthermore, if YouCompleteMe is able to look up function definition, together with a supporting docstring, it will show up in a preview window at the top of the screen:

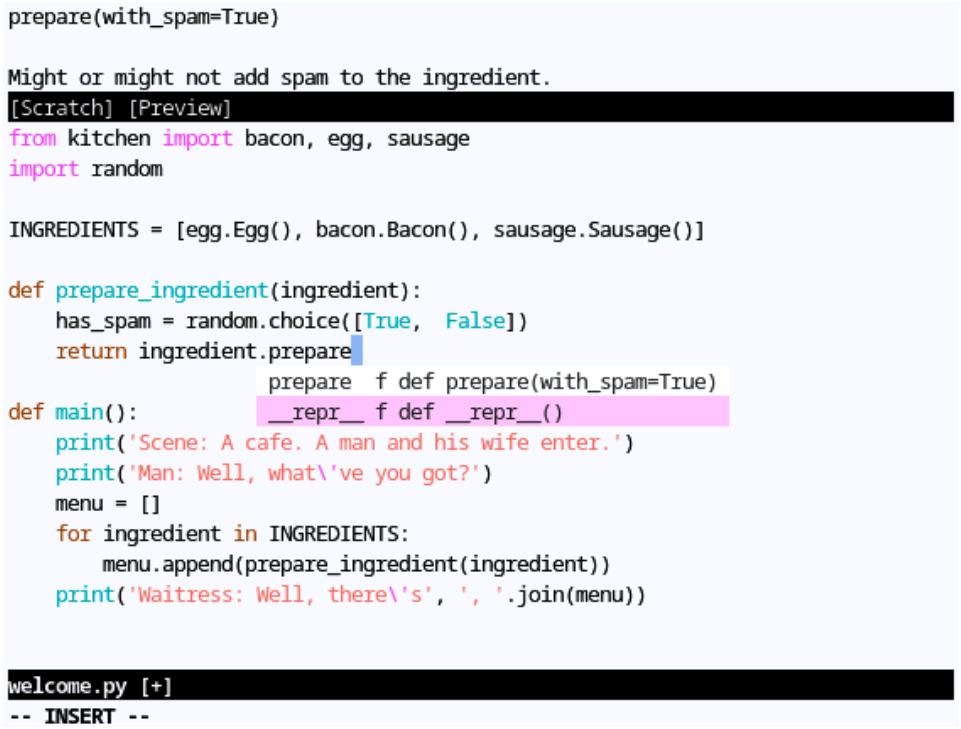

Figure 4.4 – You can see the method signature and documentation for the Ingredient.prepare method at the top of the screen

> **If you don't see the preview**
>
> The preview window only shows up when YouCompleteMe uses the semantic autocomplete engine. The semantic engine is automatically invoked when typing in insert mode (e.g., after a `.` period) or manually by pressing *Ctrl* + *spacebar*.

For Python, YouCompleteMe also allows you to jump to the function definition. Add the following mapping to your `.vimrc` file:

```
noremap <leader>] :YcmCompleter GoTo<cr>
```

Now, with the cursor over a function call, press your leader key (backslash, \, by default), followed by *]*. You will be taken to the function definition:

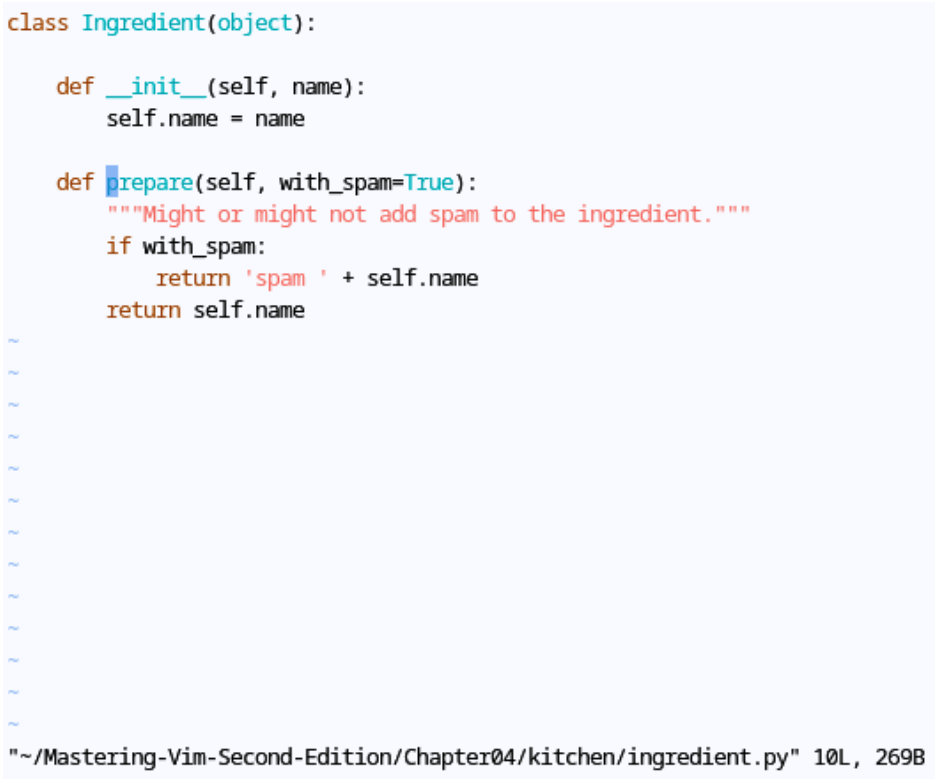

Figure 4.5 – Pressing <leader> followed by] when hovering over a method description takes you to the method definition

Alternative completion plugins

YouCompleteMe is not the only available autocomplete tool, but merely the author's favorite (and the most popular option at the time of writing this book). There are many alternatives. A quick search along the lines of *Vim autocomplete* will yield plenty of results if you're interested in an alternative.

Navigating the code base with tags

Autocomplete is helpful when writing anew, but you often have to read what somebody else wrote. A common task when navigating code bases is trying to figure out where certain methods are defined, and looking for occurrences of a certain method.

Vim has a built-in feature that allows you to navigate to the definition of a variable in the same file. With the cursor over a word, press *gd* to go to the declaration of the variable. For instance, open `welcome.py` and position your cursor at the beginning of `prepare_ingredient` on *line 17*. Press *gd*, and your cursor will jump to *line 8*, where the function is defined:

```
1 #!/usr/bin/python
2
3 from kitchen import bacon, egg, sausage
4 import random
5
6 INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]
7
8 def prepare_ingredient(ingredient):
9     has_spam = random.choice([True,  False])
10    return ingredient.prepare(has_spam)
11
12 def main():
13    print('Scene: A cafe. A man and his wife enter.')
14    print('Man: Well, what\'ve you got?')
15    menu = []
16    for ingredient in INGREDIENTS:
17        menu.append(prepare_ingredient(ingredient))
18    print('Waitress: Well, there\'s', ', '.join(menu))
19
20
21 if __name__ == '__main__':
22    main()
```

Figure 4.6 – Pressing gd takes you to the symbol definition

gd will look for a local variable declaration first. There's also *gD*, which will look for a global declaration (starting at the beginning of the file instead of the beginning of the current scope).

This feature is not syntax-aware, as out-of-the-box Vim does not know how your code is structured semantically. However, Vim supports tags—a file of semantically meaningful words and constructs across your files. For example, in Python, likely candidates for tags are classes, functions, and methods.

Exuberant Ctags

Exuberant Ctags is an external utility that generates tag files. Ctags is available at the following link: `http://ctags.sourceforge.net`.

> **Tip**
>
> If you're on a Debian-flavored distribution, you can install Exuberant Ctags by running `sudo apt install universal-ctags`.

Ctags introduces a `ctags` binary, which allows you to generate a `tags` file for your code base. Let's navigate to our project and try it out:

```
$ ctags -R .
```

This creates a `tags` file in the directory you're in.

> **Tip**
>
> You may want to set the following option in your `.vimrc` file:
>
> ```
> set tags=tags; " Look for a tags file recursively in
>  " parent directories.
> ```
>
> This will make sure that Vim looks for the `tags` file recursively in parent directories to allow you to use a single `tags` file for the whole project. The semicolon (`;`) is what tells Vim to keep looking in parent directories until a `tags` file is found.

Now, open `welcome.py` in Vim. Place your cursor over a semantically meaningful keyword, for example, the `prepare` method on *line 10*:

```
1  #!/usr/bin/python
2
3  from kitchen import bacon, egg, sausage
4  import random
5
6  INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]
7
8  def prepare_ingredient(ingredient):
9      has_spam = random.choice([True,  False])
10     return ingredient.prepare(has_spam)
11
12 def main():
13     print('Scene: A cafe. A man and his wife enter.')
14     print('Man: Well, what\'ve you got?')
15     menu = []
16     for ingredient in INGREDIENTS:
17         menu.append(prepare_ingredient(ingredient))
18     print('Waitress: Well, there\'s', ', '.join(menu))
19
20
21 if __name__ == '__main__':
22     main()
```

Figure 4.7 – Place a cursor over a method – you'll be able to move to its definition with Ctrl +]

Hit *Ctrl* + *]* to follow the tag to the definition (which is in a different file—`kitchen/egg.py`):

```
from kitchen import ingredient


class Egg(ingredient.Ingredient):

    def __init__(self):
        self.name = 'egg'

    def prepare(self, with_spam=True):
        """Becomes an omelet as you add spam!"""
        return 'spam omelet' if with_spam else self.name
~
~
~
~
~
~
~
~
~
~
~
~
"kitchen/egg.py" 11L, 264B
```

Figure 4.8 – Ctrl +] takes you to the method definition

Use *Ctrl* + *t* to go back in the tag stack (this will place your cursor back to where it was in the previous file).

> **Tip**
> Jump list navigation with *Ctrl* + *o* and *Ctrl* + *i* also works; however, the two are using different lists.

If you have multiple tags with the same name, you can cycle through the available options by using the `:tn` (**next tag**) and `:tp` (**previous tag**) commands.

You can also bring up a list of tags by using the `:ts` (**tag select**) menu. For example, if you jump to the definition of `prepare` (using *Ctrl +]* from `ingredient.prepare(has_spam)` in `welcome.py`, for instance) and execute `:ts`, you'll see the following menu:

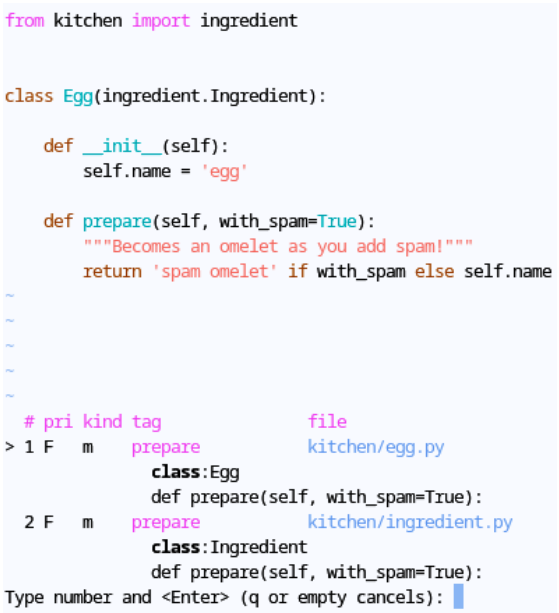

Figure 4.9 – :ts opens a list of tags, allowing you to select the definition to jump to

You can see what file, class, and method the tag refers to, and you can jump to the desired tag by entering a number.

You can also open a tag and select the menu instead of jumping to the tag under the cursor using *g]*.

You can jump to a tag location immediately as you open Vim. From your prompt, execute the following:

```
$ vim -t prepare
```

This will take you directly to the `prepare` symbol.

Automatically updating the tags

You probably don't want to have to manually run the `ctags -R .` command every time you make changes to the code. The simplest way to address this is to add the following to your `.vimrc` file:

```
" Regenerate tags when saving Python files.
autocmd BufWritePost *.py silent! !ctags -R &
```

The preceding snippet runs `ctags -R` every time you save a Python file.

You can replace the preceding `*.py` extension with different file extensions depending on the language you'd like to work with. For example, the following will generate a `tags` file for C++ files:

```
autocmd BufWritePost *.cpp,*.h silent! !ctags -R &
```

Having explored the array of code completion options and mastered the art of navigating complex code bases, we will now venture into the intriguing realm of Vim's undo tree, a game-changing feature that propels your editing capabilities to new heights.

Visualizing the undo tree

Most modern editors support an undo stack, with undoing and replaying operations. Vim takes that one step further by introducing an undo tree. If you make a change, *X*, undo it, and then make a change, *Y*—Vim still saves the change *X*. Vim supports manually browsing undo tree leaves, but there's a better way to do this.

Undotree is a plugin that visualizes the undo tree and is available from GitHub at `https://github.com/mbbill/undotree`.

> **Alternatives to Undotree**
>
> Many plugins visualize Vim's undo tree, and **Undotree** is only one of them. **Gundo** (`https://github.com/sjl/gundo.vim.git`) and its more recent fork, **Mundo** (`https://github.com/simnalamburt/vim-mundo`), are other popular choices.

> **Quickly installing plugins**
>
> If you're using `vim-plug` to manage your plugins, add the following to your `.vimrc` file: `Plug 'mbbill/undotree'`. Execute `:w | so $MYVIMRC | PlugInstall` and you'll have Mundo installed and ready to go.

Let's say you're working on `welcome.py`, with your cursor on *line 15*:

```
 1 #!/usr/bin/python
 2
 3 from kitchen import bacon, egg, sausage
 4 import random
 5
 6 INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]
 7
 8 def prepare_ingredient(ingredient):
 9     has_spam = random.choice([True,  False])
10     return ingredient.prepare(has_spam)
11
12 def main():
13     print('Scene: A cafe. A man and his wife enter.')
14     print('Man: Well, what\'ve you got?')
15     menu = []
16     for ingredient in INGREDIENTS:
17         menu.append(prepare_ingredient(ingredient))
18     print('Waitress: Well, there\'s', ', '.join(menu))
19
20
21 if __name__ == '__main__':
22     main()
```

Figure 4.10 – Place your cursor on line 15, make changes, undo them, and make a different set of changes

You're editing the highlighted line: `menu = []`. You perform the following operations:

1. Change the line to `menu = [egg.Egg()]`.
2. Undo the edit using the undo command (*u*).
3. Change the line to `menu = [bacon.Bacon()]`.

Normally, you'd expect the edit where you introduced `egg.Egg()` to be lost, but since Vim has an undo tree, the change is preserved!

Executing `:UndotreeToggle` will open two new windows in a split: the visual representation of the tree (top left) and the difference between that version and a previous snapshot (bottom left). Here's how it looks:

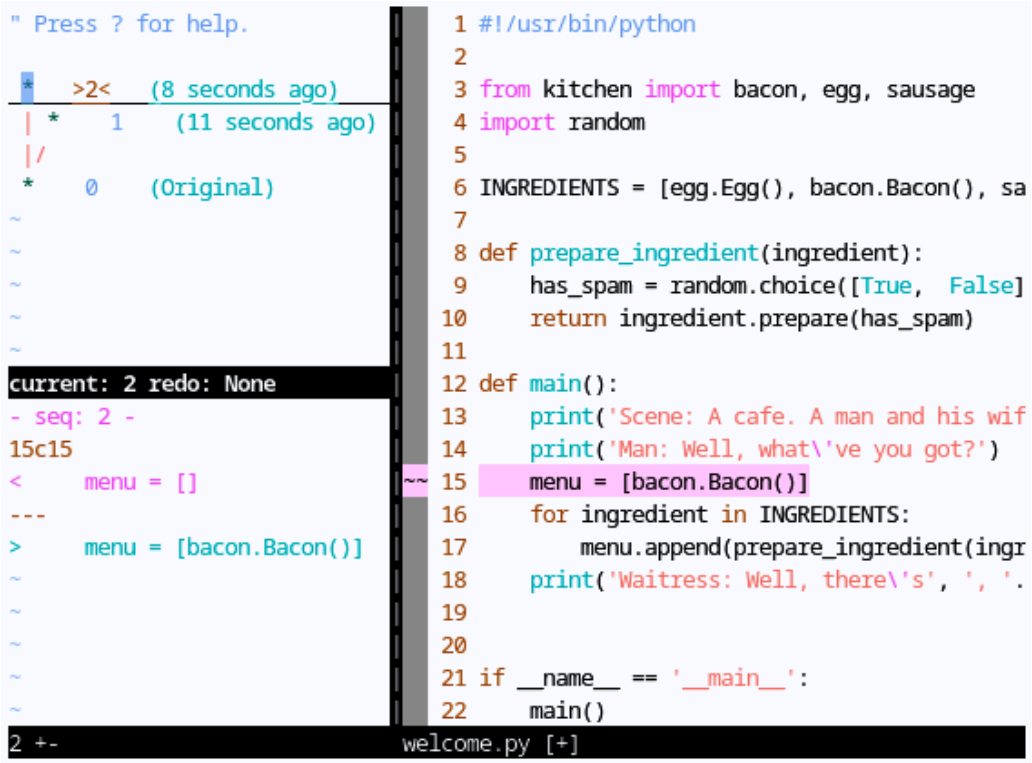

Figure 4.11 – Undotree in all its glory – the visualized tree is in the top-right window, the diff is in the bottom-left window, and the file snapshot is on the right

With your cursor over the **Undotree** window, you can navigate up and down the tree with *j* and *k* and select a revision with *Enter*. Go to the last change at the top of the tree if you're not already on it (you can use `gg` to quickly make your way to the top of the buffer). You can see how we changed the line `menu = []` to `menu = [bacon.Bacon()]` in our last change.

> **Note**
>
> The undo tree functionality exists in Vim out of the box. Plugins such as Undotree, Gundo, and Mundo merely provide a handy visualization!

Now, hit *j* to go down the tree to a different (now unused) branch (hit *Enter* to select):

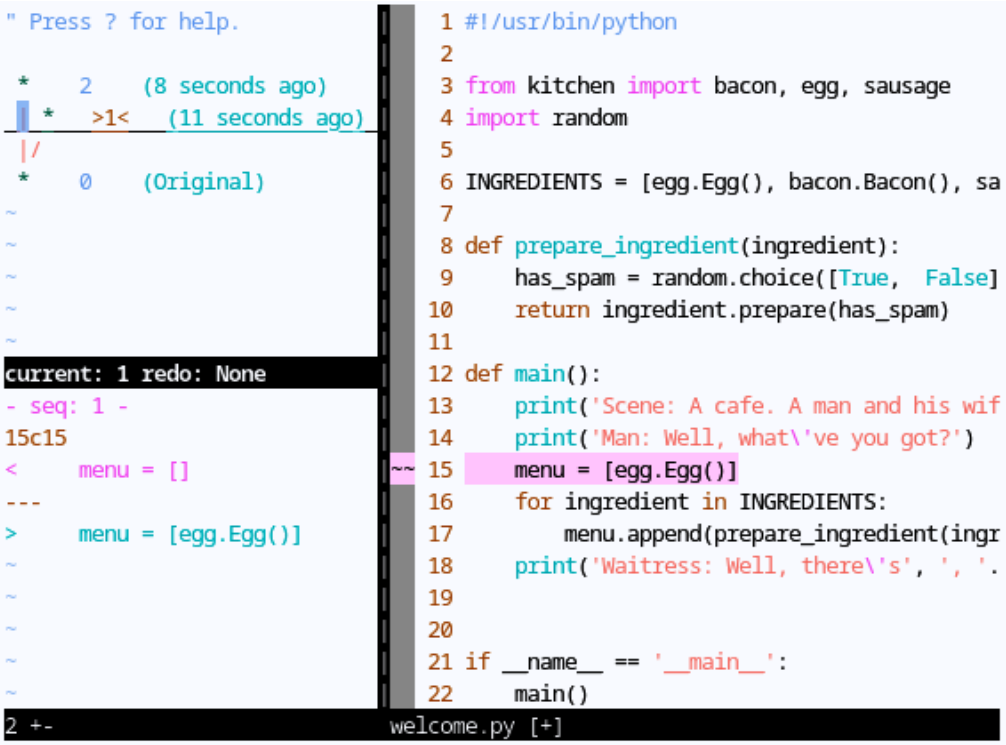

Figure 4.12 – Selecting a version from another branch in the undo tree allows you to retrieve changes that would otherwise be lost

This is the edit we thought we had lost! You can see how we replaced `menu = []` with `menu = [egg.Egg()]`. Run `:UndotreeToggle` again to hide the undo tree.

Tip

If you're anything like me, you'll use the Undotree a lot. I have it mapped to the *F5* key, to make it easier to invoke:

```
noremap <f5> :UndotreeToggle<cr> " Map Undotree to <F5>.
```

Ask for :help

If you want to learn more about the undo tree, see `:help undo-tree`. We've also covered some undo tree functionality (including persisting the undo tree between sessions) in *Chapter 1*, *Persistent Undo and Repeat*.

Summary

This chapter covered some of the advanced workflows in Vim. We looked at code autocomplete using Vim's built-in autocomplete functionality. We also looked at YouCompleteMe, a plugin that makes autocomplete syntactically aware. We also looked at Exuberant Ctags as a way to navigate more complex code bases. Lastly, we looked at Vim's undo tree (the concept) and Undotree (a plugin that makes navigating the undo tree more intuitive).

In the following chapter, we'll dive into combining Vim and version control and dealing with merge conflicts. We'll also dive into ways to build, test, and execute code in a Vim-friendly manner.

5
Build, Test, and Execute

This chapter will focus on working with version control, a fundamental component of modern development workflows, as well as building and testing code. Version control streamlines collaboration, tracks changes, and provides a reliable history of your project's evolution. You will learn to do the following:

- Working with version control (and Git in particular) if you haven't already
- Learning to productively use Git and Vim together
- Comparing and merging files with **vimdiff**
- Resolving Git conflicts using **vimdiff**
- Using **tmux**, **screen**, or Vim terminal mode to multitask and execute shell commands
- Using quickfix and location lists to capture warnings and errors
- Building and testing code using the built-in `:make` command and plugins
- Running syntax checkers manually and by using plugins

Technical requirements

Among other things, this chapter will cover working with version control. Git is the version control system of choice for this chapter; however, the lessons learned are applicable across different systems. A section is dedicated to a quick-and-dirty introduction, but if you want to get the most out of version control systems, you may want to read up on the version control system of your choice.

Throughout this chapter, we'll be making changes to our `.vimrc` file. You can make these changes as you go, or download them from GitHub at `https://github.com/PacktPublishing/Mastering-Vim-Second-Edition/tree/main/Chapter05`. Git installation and configuration instructions are also included in the repository.

Working with version control

This section will illustrate working with **version control systems** (often abbreviated to **VCS**) by using Git as an example.

> **VCS landscape may change**
> Git seems to be the most popular version control system at the time of writing this book. Suggestions in this section can be applied to whichever version control system you choose to work with (or, more likely, whichever VCS you're locked into).

Modern development is nearly impossible without version control, and if you're working with code, it's more than likely you'll have to deal with it. This section gives you a refresher on how to use one of the most popular version control systems today—Git. We then cover how to work with Git from within Vim to make Git commands more robust and interactive.

A quick-and-dirty version control and Git introduction

You can safely skip this section if you're comfortable with Git.

Git lets you track a history of changes to files and helps ease the pain of multiple people working on the same set of files. Git is a distributed version control system, meaning every developer owns a mirrored copy of the code base on their system.

If you're on a Debian-flavored Linux distribution, you can install Git by running the following:

```
$ sudo apt install git
```

If you're on a different system, you can download the binaries or find more instructions from `git-scm.com/download`. You'll want to configure your username and your email address:

```
$ git config --global user.name 'Your Name'
$ git config --global user.email 'your@email'
```

You're now ready to use Git! If you find yourself stuck, Git has an extensive help system (in addition to a set of tutorials on `git-scm.com`):

```
$ git help
```

Concepts

Git represents a history of changes to files using commits—atomic sets of changes to files. In addition to a diff of changes, each commit has a (hopefully) descriptive message attached to it (by the author of the commit), allowing you to determine what changes were made at any given point in time.

Commit history is not just linear and can branch, allowing Git users to work on multiple features without stepping on each other's toes. For example, in the following example (read from bottom to top), **Lobster Thermidor** was built in a `master` (main) branch, while **Tomato** was developed in parallel in its own branch, called `feature-tomato`:

```
* Merged feature-tomato into the master branch
|\
* | Improved Lobster Thermidor recipe
| * Included tomato in the menu
| * Added tomato class (feature-tomato branch)
|/
* Added Lobster Thermidor to the menu
* Initial commit (master branch)
```

Git is a distributed version control system, meaning that there is no central place to talk to: every developer owns a full copy of the repository.

Setting up a new project

In this example, we'll be working with `Chapter05/spam/` from `https://github.com/PacktPublishing/Mastering-Vim-Second-Edition/tree/main/Chapter05/spam`. You can also follow along with any project you'd like. Follow these steps if you're setting up a new Git repository:

1. Initialize the Git repository in the project's root directory:

```
$ cd Chapter05/spam
$ git init
```

2. Stage all files in a directory to be added to the initial commit:

```
$ git add .
```

3. Create an initial commit:

```
$ git commit -m "Initial commit"
```

Here's a sample output from the previous commands:

```
~/Mastering-Vim-Second-Edition/Chapter05/spam$ git init
Initialized empty Git repository in /home/ruslano/Mastering-Vim-Second-Edition/C
hapter05/spam/.git/
~/Mastering-Vim-Second-Edition/Chapter05/spam$ git add .
~/Mastering-Vim-Second-Edition/Chapter05/spam$ git commit -m "Initial commit"
[master (root-commit) e2c63ef] Initial commit
 5 files changed, 57 insertions(+)
 create mode 100644 kitchen/bacon.py
 create mode 100644 kitchen/egg.py
 create mode 100644 kitchen/ingredient.py
 create mode 100644 kitchen/sausage.py
 create mode 100644 welcome.py
~/Mastering-Vim-Second-Edition/Chapter05/spam$
```

Figure 5.1 – Initializing the Git repository within the Chapter05/Spam directory.

You should now be ready to work with your newly created repository.

If you want to have your repository backed up somewhere outside of your machine, you may want to use a service such as GitHub. See `github.com/new` to create a new repository, and add the repository URL to your project (where `<url>` needs to be replaced with the repository URL—something like `https://github.com/<your-username>/spam.git`):

```
$ git remote add origin <url>
```

Now, you just need to push the changes from your local repository:

```
$ git push -u origin master
```

To keep the repositories in sync, you'll have to push every time you add a new commit; see the *Working with Git* section for details.

Cloning an existing repository

If you already have code in a remote repository (for example, on GitHub), all you need to do is "clone" it—make a local copy. Find the repository URL, either over HTTPS (for example, `https://github.com/vim/vim.git`) or SSH (for example, `git@github.com:vim/vim.git`). Execute the following command (replacing `<url>` with the repository URL):

```
$ git clone <url>
```

This should download the repository to a directory with the project name on your machine.

Your local and remote repositories will now operate independently. If you want to update your local repository with changes from the remote repository (the one you cloned), you'll have to run `git pull --rebase`.

Working with Git

Git is rather extensive, but here are some basic commands to get you started. Let's work in our newly created repository, `spam/`.

Adding files, committing, and pushing

Let's add a file to our repository, `kitchen/lobster_thermidor.py`:

```
"""Lobster Thermidor."""

from kitchen import ingredient


class LobsterThermidor(ingredient.Ingredient):

    def __init__(self):
        self.name = 'Lobster Thermidor'
```

Next, let's add a bit to `welcome.py` that invokes the file (the changes are highlighted in bold):

```
...
from kitchen import bacon, egg, lobster_thermidor, sausage
...
INGREDIENTS = [
    egg.Egg(),
    bacon.Bacon(),
    lobster_thermidor.LobsterThermidor(),
    sausage.Sausage()]
...
```

You can check the status of your files (to see which changes are going to make it into a commit) by running the following command:

```
$ git status
```

The output of the command will show you that you modified `welcome.py` and added `kitchen/lobster_thermidor.py`:

```
~/Mastering-Vim-Second-Edition/Chapter05/spam$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   welcome.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        kitchen/lobster_thermidor.py

no changes added to commit (use "git add" and/or "git commit -a")
~/Mastering-Vim-Second-Edition/Chapter05/spam$
```

Figure 5.2 – Output of `git status` shows that `welcome.py` was modified, and `kitchen/lobster_thermidor.py` was added.

Whenever you want to save your files in history, you can stage the files. You can do so individually by executing the following:

```
$ git add <filename>
```

Alternatively, you can stage all of the files at once by running the following:

```
$ git add .
```

If you run `git status`, you'll see that the files are now staged to be committed:

```
~/Mastering-Vim-Second-Edition/Chapter05/spam$ git add .
~/Mastering-Vim-Second-Edition/Chapter05/spam$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   kitchen/lobster_thermidor.py
        modified:   welcome.py

~/Mastering-Vim-Second-Edition/Chapter05/spam$
```

Figure 5.3 – The `git status` output shows that `kitchen/lobster_thermidor.py` and `welcome.py` are staged to be committed.

To commit a file or a set of files to history, execute the following:

```
$ git commit -m "<informative message describing the changes>"
```

For instance, here's how we would commit our changes to `kitchen/lobster_thermidor.py` and `welcome.py`:

```
~/Mastering-Vim-Second-Edition/Chapter05/spam$ git commit -m "Added Lobster Therm
idor to the menu"
[master 287ac1b] Added Lobster Thermidor to the menu
 2 files changed, 15 insertions(+), 2 deletions(-)
 create mode 100644 kitchen/lobster_thermidor.py
~/Mastering-Vim-Second-Edition/Chapter05/spam$
```

Figure 5.4 – The output of the `git commit` command.

To push your changes to a remote repository (if you created one earlier), run the following:

```
$ git push
```

To synchronize your changes with the changes other people made, run the following (in fact, it's usually wise to pull before pushing if you work with multiple people):

```
$ git pull --rebase
```

Now, if you want to view the commit history, run the following:

```
$ git log
```

Here's how the `git log` output looks for our project so far:

```
commit 287ac1b01c363dcc97e6410660f4a9c4d1117ce0 (HEAD -> master)
Author: Ruslan Osipov <ruslan@rosipov.com>
Date:   Sun Jan 7 15:26:53 2024 -0800

    Added Lobster Thermidor to the menu

commit e2c63ef73f1c83d88d05f672bdf6e94031f66672
Author: Ruslan Osipov <ruslan@rosipov.com>
Date:   Fri Dec 29 11:17:15 2023 -0800

    Initial commit
(END)
```

Figure 5.5 – The output of the `git log` command shows two commits we made earlier.

> **Note**
>
> You might have spotted that the commits span a winter holiday break. Take good care of yourself, and make time for yourself and your family!

At the top, we can see a commit we just created. Right below is the initial commit.

If you'd like to pull a particular commit into your working copy (for example, to see how things were at the initial commit), run the following (where `<sha1>` is the alphanumeric commit ID, for example, `e2c63ef73f1c83d88d05f672bdf6e94031f66672` for `Initial commit` in the previous screenshot):

```
$ git checkout <sha1>
```

Creating and merging branches

Separate branches are often used to create separate chunks of work. Once the feature is ready, the branches are merged back into the master (primary) branch. To create a new branch, run the following:

```
$ git checkout -b <branch-name>
```

For instance, we could create a branch in which we add a new animal type:

```
$ git checkout -b feature-tomato
```

It will appear as the following:

```
..Chapter05/spam$ git checkout -b feature-tomato
Switched to a new branch 'feature-tomato'
..Chapter05/spam$
```

Figure 5.6 – `git checkout -b` creates a new branch and makes the newly created branch active.

Now, you can perform work on this branch as usual. For instance, we could add a new `kitchen/tomato.py` and modify `welcome.py` just like we did in the previous section:

```
..Chapter05/spam$ vim kitchen/tomato.py
..Chapter05/spam$ git add kitchen/tomato.py
..Chapter05/spam$ git commit -m "Add a Tomato class"
[master ada936e] Add a Tomato class
 1 file changed, 7 insertions(+)
 create mode 100644 kitchen/tomato.py
..Chapter05/spam$ vim welcome.py
..Chapter05/spam$ git add welcome.py
..Chapter05/spam$ git commit -m "Added tomato to the spam sketch"
[master 9ad9656] Added tomato to the spam sketch
 1 file changed, 3 insertions(+), 2 deletions(-)
..Chapter05/spam$
```

Figure 5.7 – I captured adding the `Tomato` class and including it in `welcome.py` in two separate commits.

Now that our feature is ready (we've added the leopard), we're ready to merge the `feature-tomato` branch back into the `master` (primary) branch. To see a list of all branches, run the following:

```
$ git branch -a
```

The branch you are currently on is marked with an asterisk (`*`):

```
..Chapter05/spam$ git branch -a
* feature-tomato
  master
..Chapter05/spam$
```

Figure 5.8 – `git branch -a` displays two branches: `feature-tomato` and `master`. The currently active branched is marked with an asterisk (`*`).

To move to another branch, run the following:

```
$ git checkout <branch-name>
```

In our case, let's move to the `master` branch:

```
$ git checkout master
```

Now, we just need to merge our feature branch into the branch we're currently on:

```
$ git merge feature-tomato
```

A helpful message will display the result of the merge:

```
..Chapter05/spam$ git checkout master
Switched to branch 'master'
..Chapter05/spam$ git merge feature-tomato
Updating 287ac1b..8b3810b
Fast-forward
 kitchen/tomato.py | 7 +++++++
 welcome.py        | 5 +++--
 2 files changed, 10 insertions(+), 2 deletions(-)
 create mode 100644 kitchen/tomato.py
..Chapter05/spam$
```

Figure 5.9 – `git merge` command merges specified branch (`feature-tomato`) into the branch you are currently on (`master`).

> **Tip**
>
> If your repository is in GitHub, don't forget to run `git push` after you're done to propagate your changes to the remote repository.

Now that you're armed with surface-level familiarity with git (or your memory has been refreshed), let's talk about how Vim and Git can play together nicely.

Integrating Git with Vim (vim-fugitive)

This section assumes that you understand the basics of working with Git. If you don't (or it's been a while), see the previous *Quick and dirty version control and Git introduction* section.

Tim Pope's **vim-fugitive** is a plugin that makes sure you don't need to leave Vim to interact with Git. Since you're editing the files in Vim, you might as well take care of dealing with version control of said edits in the editor. The plugin is available from `https://github.com/tpope/vim-fugitive`.

> **Installing vim-fugitive**
>
> If you're using **vim-plug**, you can install **vim-fugitive** by adding `Plug 'tpope/vim-fugitive'` to your `.vimrc` file and running `:w | source $MYVIMRC | PlugInstall`.

A lot of the commands that **vim-fugitive** provides are a mirror of external Git commands. However, the output is often a lot more interactive. The main vim-fugitive command is `:Git` – you can follow it with a command you already know, or run without arguments to get an interactive status output. Give it a shot:

```
:Git
```

You'll see the familiar `git status` output in a split window (you may want to make some changes to the source files without committing them to have some `git status` output to work with):

```
Head: master
Help: g?

Untracked (1)
? tags

Unstaged (1)
M welcome.py
~
~
~
<ugitive:///home/ruslano/Mastering-Vim-Second-Edition/Chapter05/spam/.git// [RO]
        bacon.Bacon(),
        lobster_thermidor.LobsterThermidor(),
        sausage.Sausage(),
        tomato.Tomato()]

def prepare_ingredient(ingredient):
    return ingredient.prepare(has_spam=True)

def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
welcome.py
:Git
```

Figure 5.10 – The `:Git` command (or simply `:G`) opens an interactive status buffer.

Unlike the `git status` output, this window is interactive. Move your cursor over one of the files (parentheses, *(* and *)*, allow you to cycle through files as well). Try some of the supported commands:

- - will stage or unstage the file
- *cc* or `:Git commit` will commit the staged files
- *dd* or `:Git diff` will open a diff
- *g?* displays help with more commands

`:Git log` opens a history of commits related to the currently open file:

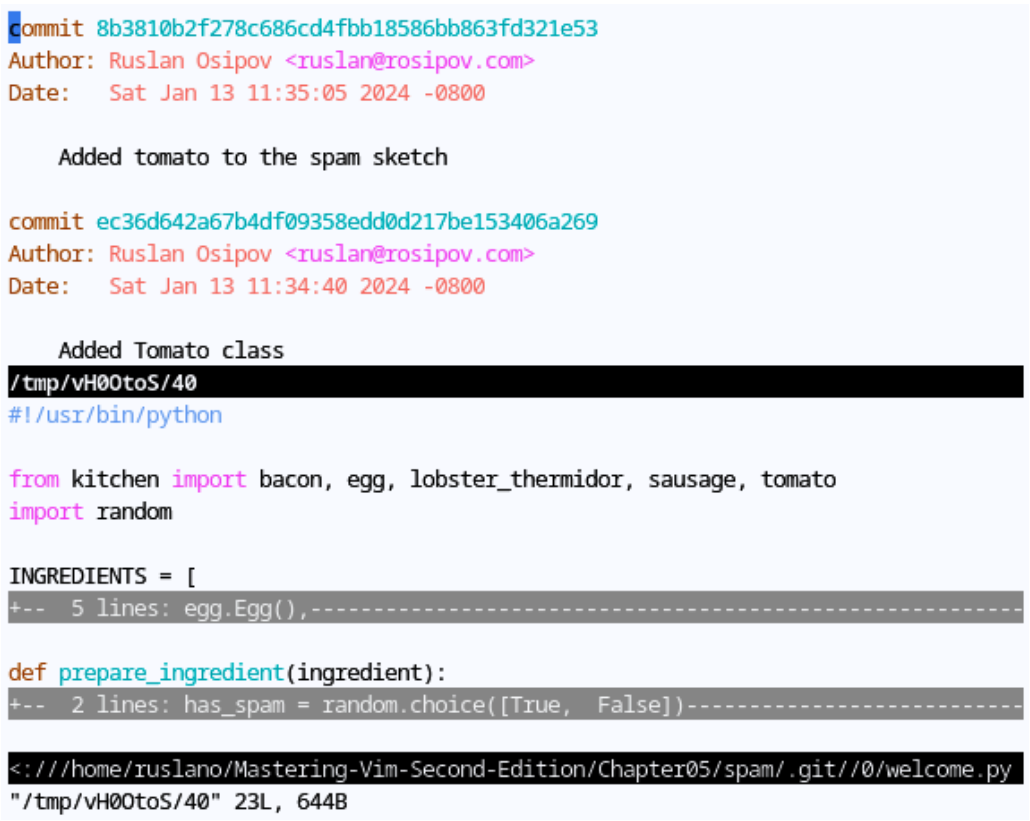

Figure 5.11 – `:Git log` opens `git log` output in a split window (how shocking!).

`git blame` is a command that lets you quickly figure out who changed every line of the file and when. This way, you can blame other developers (or, most often, yourself in the past) for bugs in your code! `:Git blame` displays interactive `git blame` output in a vertical split window:

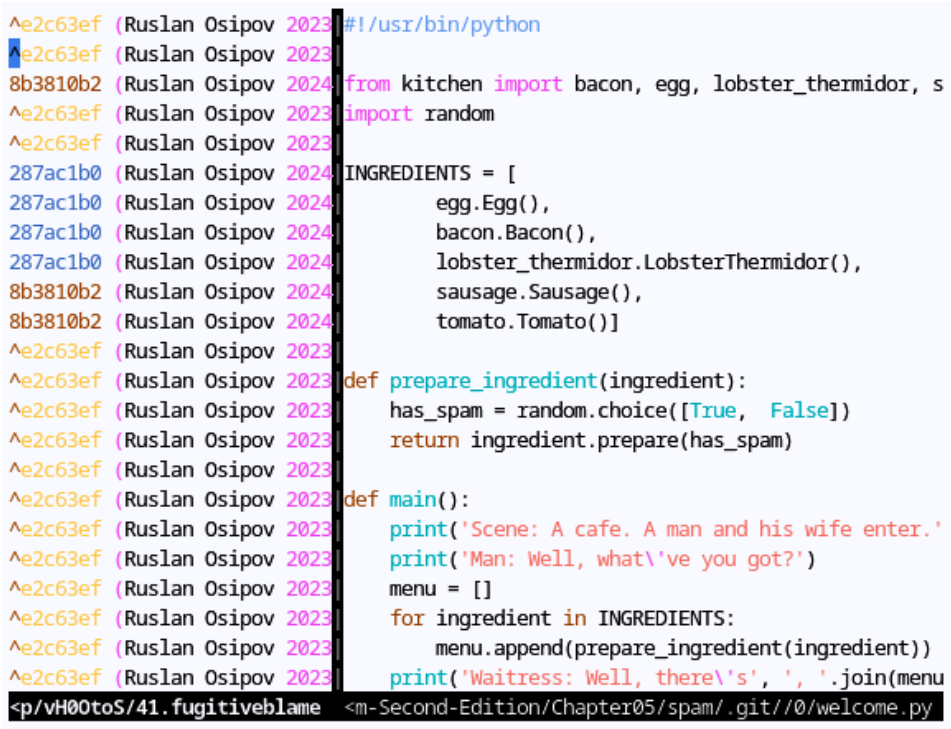

Figure 5.12 – `:Git blame` opens interactive `git blame` output in a split window.

`:Git blame` displays the relevant commit ID, name of the commit author, and commit date and time (hidden in the screenshot) next to each line in the file.

Some useful shortcuts for `:Git blame` are as follows:

- `C`, `A`, and `D` resize the blame window up until the commit, author, and date, respectively
- *Enter* opens a diff of the chosen commit
- *o* opens a diff of the chosen commit in a split window
- *g?* displays help with more commands

`:Git blame` is an extremely useful tool for figuring out when things went wrong.

There are even more really handy wrappers provided in this tool, such as the following:

- `:Gread` checks out the file straight into a buffer for a preview
- `:Ggrep` wraps around `git grep` (Git provides a powerful `grep` command that lets you search through tracked files at any moment in time—see `https://git-scm.com/docs/git-grep` for details)
- `:GMove` moves the files (while renaming the buffers)
- `:GDelete` wraps `git remove` commands

Don't forget to use Vim help (for example, `:help fugitive`) to learn more about the plugins!

> **No really, get some :help**
>
> You're reading a book, which has the unfortunate property of being stuck in time. Plugin authors, including Tim Pope, author of vim-fugitive, update and improve their plugins often. Commands might change (in fact, vim-fugitive changed significantly between different editions of this book), and more powerful commands can be added. Read **:help**, and check `README.md` in a plugin's repository – you never know what you might find.

Since you're now familiar with Git and its integration with Vim, let's dive into an area where Vim truly shines: resolving merge conflicts.

Resolving conflicts with vimdiff

Often, during development, you'll find yourself needing to compare some files—be it comparing different output or versions of a file, or dealing with merge conflicts as multiple developers collaborate on a single file. Vim provides `vimdiff`, a standalone binary that excels at file comparison operations.

Comparing two files

Using `vimdiff` to compare two files is fairly simple. Let's look at two files in `spam/kitchen/`: `kitchen/bacon.py` and `kitchen/egg.py`. We'd like to know what's different between the two.

> **Code location**
>
> The files from this example are available from `https://github.com/PacktPublishing/Mastering-Vim-Second-Edition/tree/main/Chapter05/spam`.

Open the files with `vimdiff`:

```
$ vimdiff kitchen/bacon.py kitchen/egg.py
```

You will be greeted by the following screen (how colorful the screen is will depend on your `colorscheme`):

Figure 5.13 – The `vimdiff` window shows the differences between `kitchen/bacon.py` and `kitchen/egg.py`.

You can see `kitchen/bacon.py` open in a split on the left and `kitchen/egg.py` in a split on the right. Different lines are highlighted in one color, and different characters are highlighted in another color.

You can navigate from one change to another by using `]c` to move forward and `[c` to move backward.

You can pull and push changes from one file to another:

- *do* or `:diffget` (**do** stands for diff obtain) will move the change to the active window
- *dp* or `:diffput` (**dp** stands for diff put) will push the change from the active window

> **Tip**
> If you want to copy the content of a whole file from one file into another, you can use the `:%diffget` or `:%diffput` commands.

For example, if you want to pull the `class Bacon` line from `kitchen/bacon.py` into `kitchen/egg.py`, you would navigate to the desired change using `] c` and press *do* to obtain it. The highlighting will disappear and the `kitchen/egg.py` buffer will now contain the desired change:

Figure 5.14 – *do* obtains diff for the given chunk into the selected buffer.

> **Tip**
>
> `vimdiff` automatically updates the highlighting if you're using `:diffget` and `:diffput` to move changes between files. If you edit the files manually, you'll have to update the highlighting by running `:diffupdate`, or `:diffu` for short.

You can diff as many files as you'd like at the same time; however, you won't be able to use the *do* and *dp* shortcuts. Let's open three files with `vimdiff`:

```
$ vimdiff kitchen/bacon.py kitchen/egg.py kitchen/sausage.py
```

You can see all three side by side:

Figure 5.15 – **vimdiff** can diff multiple files at once. Here, you can see diffs between `kitchen/bacon.py`, `kitchen/egg.py`, and `kitchen/sausage.py`.

Since there are multiple buffers now, you'll have to specify which buffer you would like to get the changes to or from. Both `:diffget` and `:diffput` (which can be shortened to `:diffg` and `:diffp`) take the buffer specification argument, which can either be a buffer number (look it up from `:ls!`) or a partial buffer name.

For example, while in the `kitchen/bacon.py` window, you can push the change under the cursor into `kitchen/egg.py` by running the following:

```
:diffput egg
```

The desired line will be pushed into the `kitchen/egg.py` buffer:

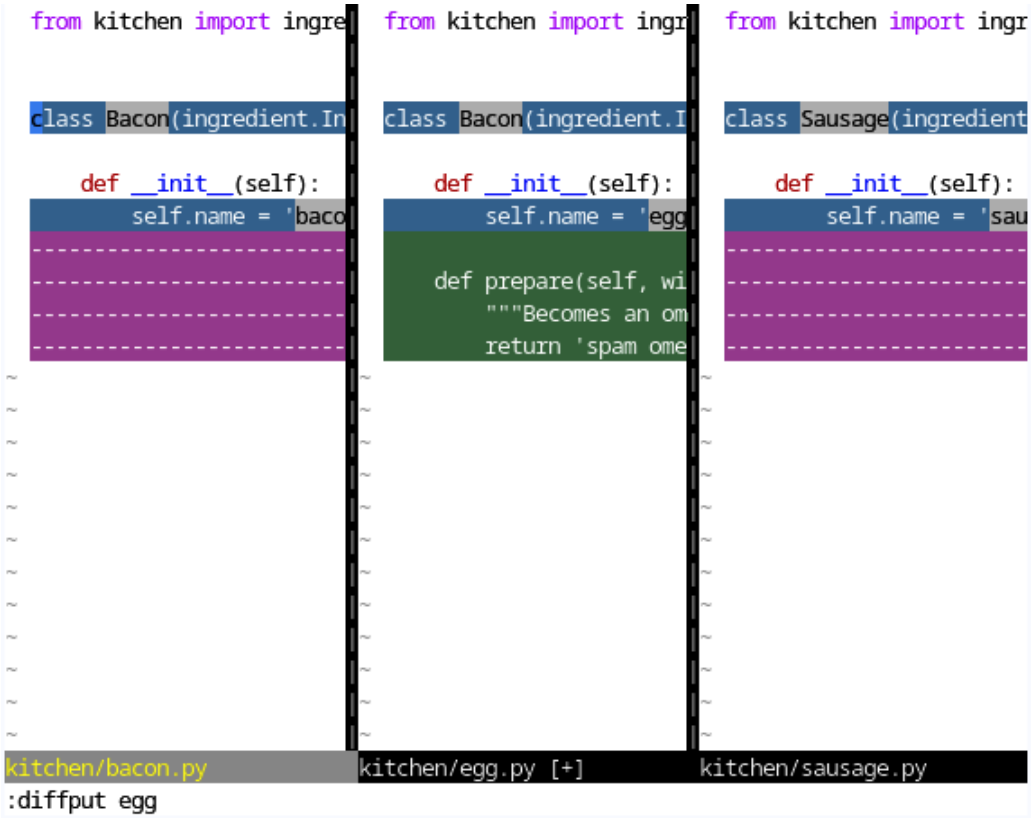

Figure 5.16 - `:diffput <partial buffer name>` will put the change under the cursor from the selected buffer into the target buffer.

> **Tip**
>
> If you work with `vimdiff` a lot, don't forget to take the time to make useful aliases and key bindings, as demonstrated in *Chapter 3*, *Follow the Leader!* The default ones are rather long.

vimdiff and Git

Using `vimdiff` as a Git merge tool can be pretty confusing—Vim bombards you with four windows, a number of keywords, and not a lot of explanation.

git config

First and foremost, configure Git to use `vimdiff` as a merge tool:

```
$ git config --global merge.tool vimdiff
$ git config --global merge.conflictstyle diff3
$ git config --global mergetool.prompt false
```

This will set Git as the default merge tool, will display a common ancestor while merging, and will disable the prompt asking you to open `vimdiff`.

Creating merge conflict

Let's use the `spam/` repository we initialized earlier in this chapter as an example (or you can follow along with your own example if you have a merge conflict you're trying to resolve):

```
$ cd spam/
```

We'll create an additional branch that will conflict with the `master` branch. In the `feature-no-spam` branch, we'll change the `prepare_ingredient` method to not include spam in any dishes. In the meantime, we'll update the `master` branch to always include spam in every dish.

The order of operations is important in creating the conflict, so you may want to follow closely.

We can start by creating a branch and editing `welcome.py`:

```
$ git checkout -b feature-no-spam
$ vim welcome.py
```

Let's update the `prepare_ingredient` method to exclude spam from every dish. We'll do that by hardcoding `with_spam=False`:

```
...
def prepare_ingredient(ingredient):
    return ingredient.prepare(with_spam=False)
...
```

Now, commit the change:

```
$ git add welcome.py
$ git commit -m "Exclude spam from every dish"
```

We can now switch back to the master branch to make our next set of changes:

```
$ git checkout master
$ vim welcome.py
```

This time, we'll update the `prepare_ingredient` method to include spam in every dish instead:

```
...
def prepare_ingredient(ingredient):
    return ingredient.prepare(with_spam=True)
...
```

Commit the file:

```
$ git add welcome.py
$ git commit -m "Include spam in every dish"
```

It's time to merge the `feature-no-spam` branch with `master`:

```
$ git merge feature-no-spam
```

Uh-oh! A merge conflict:

```
~/Mastering-Vim-Second-Edition/Chapter05/spam$ git merge feature-no-spam
Auto-merging welcome.py
CONFLICT (content): Merge conflict in welcome.py
Automatic merge failed; fix conflicts and then commit the result.
~/Mastering-Vim-Second-Edition/Chapter05/spam$
```

Figure 5.17 – `git merge` failed due to merge conflicts.

Resolving a merge conflict

Start the Git merge tool (which is `vimdiff`, since we configured it earlier):

```
$ git mergetool
```

You will be treated to quite a light show, with four windows and a lot of colors thrown at you:

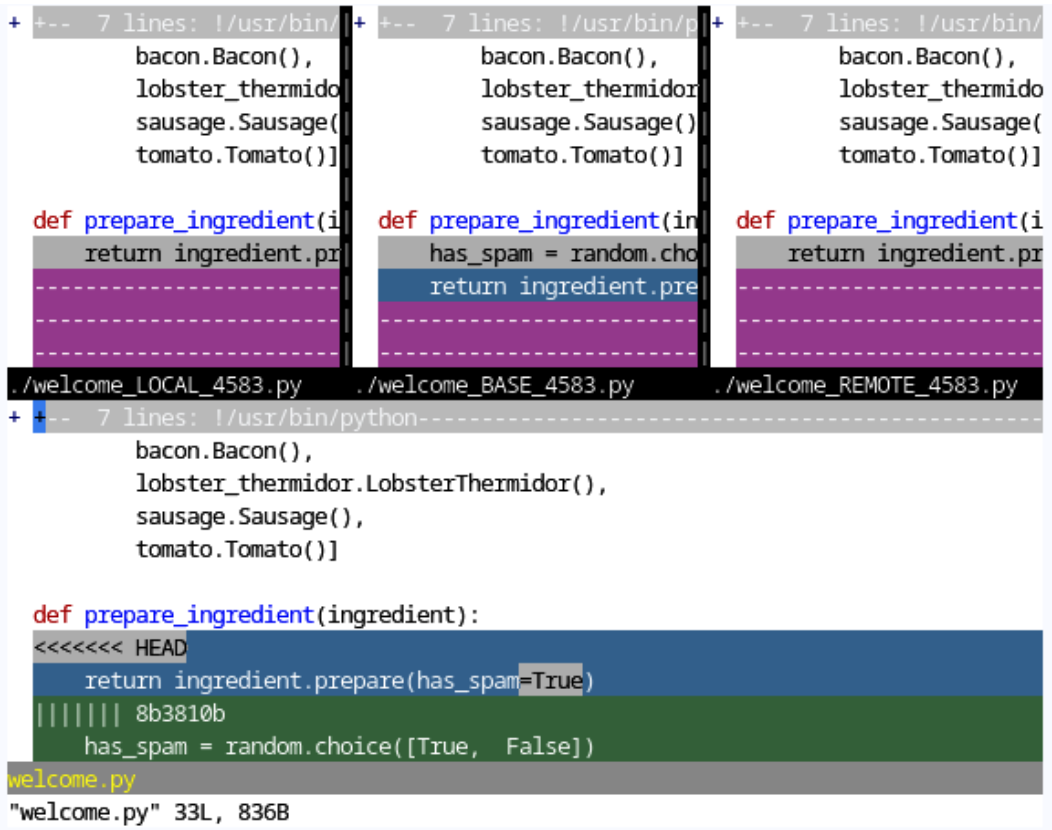

Figure 5.18 – **vimdiff** as `git mergetool` looks terrifying, but give it a moment and it'll make sense.

It's okay to be scared, but it's not as terrifying as it looks.

Local changes (the master branch in this case) are in the upper-left window, followed by the closest common ancestor and the `feature-no-spam` branch in the upper-right corner. The result of the merge is in the bottom window.

Let's get into more detail, from left to right, top to bottom:

- **LOCAL**: This is a file from the current branch (or whatever you're merging into)
- **BASE**: The common ancestor—how the file looked before both changes took place
- **REMOTE**: The file you are merging from another branch (`feature-no-spam` in this case)
- **MERGED**: The merge result—this is what gets saved as output

In the **MERGED** window, you'll see conflict markers. You don't need to interact with them directly, but it's good to have a vague idea of what they mean. Conflict markers are identified by `<<<<<<<` and `>>>>>>>`:

```
<<<<<<< [LOCAL commit/branch]
[LOCAL change]
||||||| merged common ancestors
[BASE - closest common ancestor]
=======
[REMOTE change]
>>>>>>> [REMOTE commit/branch]
```

Since we have multiple files, simply running `do` (`:diffget`) or `dp` (`:diffput`) without arguments would not be enough.

Assuming you want to keep the `REMOTE` change (from the `feature-no-spam` branch), move your cursor to the window with the `MERGED` file (the one at the bottom). Now, move the cursor to the next change (in addition to regular movement keys, you can use `]c` and `[c` to move by changes). Now, execute the following:

```
:diffget REMOTE
```

This will get the change from the `REMOTE` file and place it into the `MERGED` file. You can shorten these commands:

- Get a `REMOTE` change using `:diffg R`
- Get a `BASE` change using `:diffg B`
- Get a `LOCAL` change using `:diffg L`

Repeat for every conflict. Once you're done addressing the conflicts, save the `MERGED` file and exit `vimdiff` (running `:wqa` would be the fastest way) to complete the merge (or move on to the next file if you have more conflicts).

> **Tip**
>
> Merge conflicts tend to leave `.orig` files in your working directory (for example, `welcome.py.orig`); feel free to discard those once you're done merging.

Don't forget to commit the merge results (`git commit -m "Fixed a pesky merge conflict"`) once you're done.

Configuring diff behavior

Just like the rest of Vim, vimdiff is highly customizable. Read up on `:help 'diffopt'` to learn how you can change diff behavior (e.g., to make diff case insensitive or ignore whitespace changes).

tmux, screen, and Vim terminal mode

Software development often involves more than just writing code: executing your binaries, running tests, and using command-line tools to accomplish certain tasks. That's where session and window managers come in.

Modern desktop environments allow you to have multiple windows, but we'll focus on how you can manage the tasks you need to accomplish in a single Terminal session.

tmux

tmux is a Terminal multiplexer: it allows you to manage multiple Terminal windows on a single screen.

Tip

If you're on a Debian-based distribution, you can install tmux using `sudo apt install tmux`. You can also build tmux from source, which is available from GitHub: `https://github.com/tmux/tmux`.

You can start it by invoking `tmux` in the Terminal:

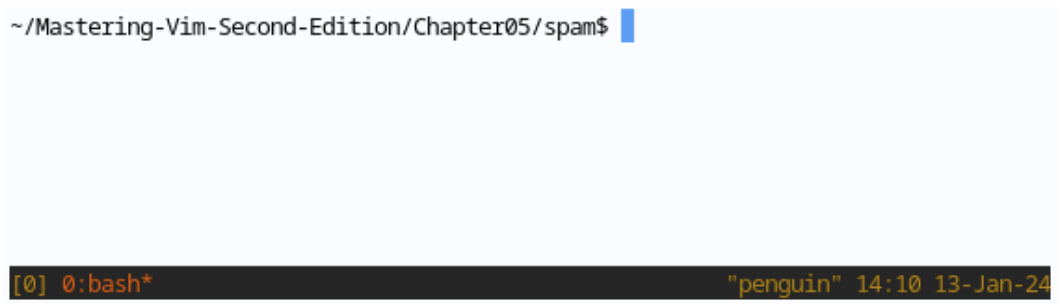

Figure 5.19 – A **tmux** window.

Panes are just like splits

tmux allows you to have multiple panes (the equivalent of windows in Vim) and windows (the equivalent of tabs). To access tmux functionality, you first need to hit a prefix key, followed by a command. The default prefix key is *Ctrl* + *b*.

Customize tmux to your liking

You can rebind the default prefix key to something else by creating or editing your `~/.tmux.conf` file. For example, if you wanted to use *Ctrl* + \ as a prefix instead of *Ctrl* + *b*, you would add the following:

```
# Use Ctrl-\ as a prefix.
unbind-key C-b
set -g prefix 'C-\'
bind-key 'C-\' send-prefix
```

Restart tmux (or execute *Ctrl* + *b* followed by `:source-file ~/.tmux.conf`) to apply the configuration.

To split the screen horizontally, use *Ctrl* + *b* followed by ":

Intuitive key bindings

For some reason, I could never get used to the default bindings. I remember the pipe character, `|`, being a lot easier for creating vertical splits. My `~/.tmux.conf` file contains the following:

```
# Use | to create vertical splits.
bind | split-window -h
unbind '"'
```

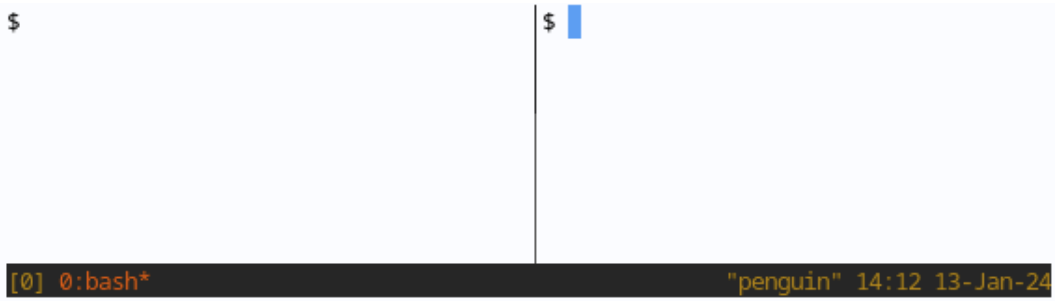

Figure 5.20 – A **tmux** window with a vertical split (the default key binding: *Ctrl* + *b*, *%*).

To create a vertical split, hit *Ctrl* + *b*, followed by *%*:

Intuitive key bindings

The same as with horizontal splits, I find a hyphen, `-`, to be a lot easier to remember for creating horizontal split windows. My `~/.tmux.conf` file contains the following:

```
# Use - to create horizontal splits.
bind - split-window -h
unbind '%'
```

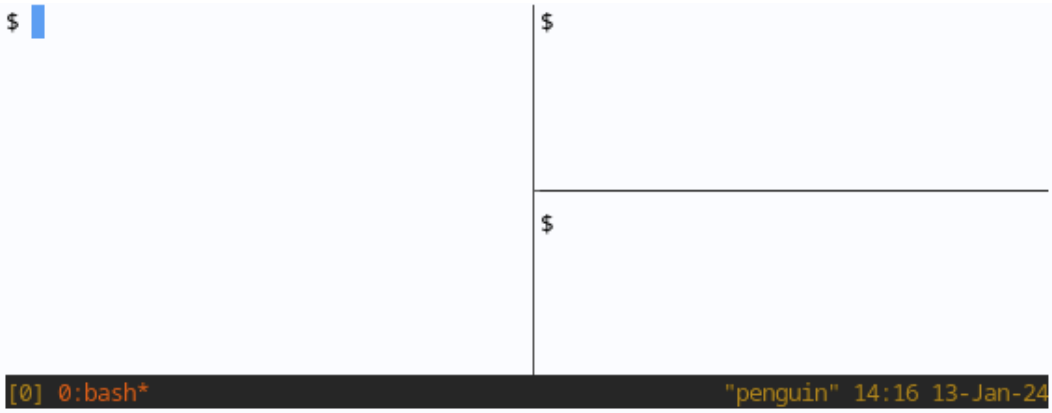

Figure 5.21 – A **tmux** window with a vertical and a subsequent horizontal split (the default key binding for a horizontal split: *Ctrl* + *b*, *"*).

You can navigate the panes by using *Ctrl* + *b*, followed by an arrow key. Every pane operates independently, and you can change directories, execute commands, and (most importantly) use Vim in each one of the panes.

Use hjkl to move around!

If you're already used to `hjkl`, arrow key navigation might feel unwieldy. Add the following to your `~/.tmux.conf` file to add `hjkl` movement support:

```
bind h select-pane -L
bind j select-pane -D
bind k select-pane -U
bind l select-pane -R
```

In the following example, I have a file with some code loaded into the left-hand pane; I'm editing `.vimrc` in the upper-right pane and listing files with `ls` in the lower-right pane:

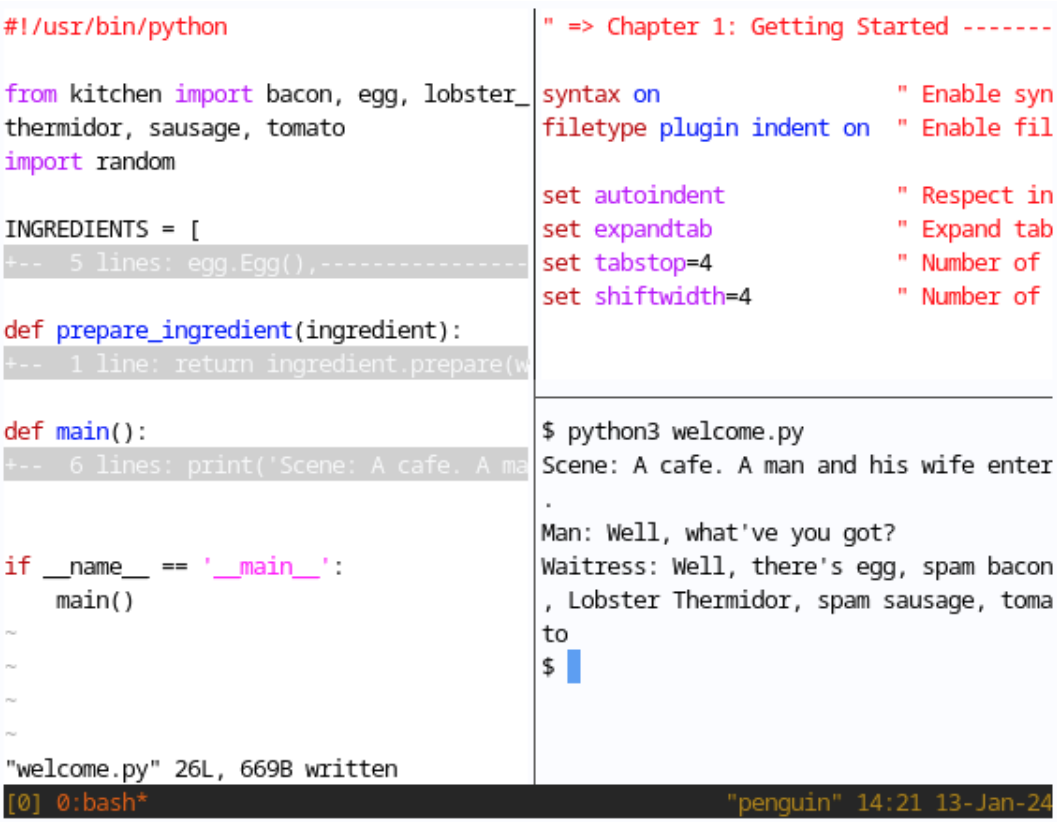

Figure 5.22 – **tmux** with `welcome.py` open in Vim in the left pane, `.vimrc` in the top-right pane, and a command line executing `python3 welcome` in the bottom-right pane.

Exit the session by executing `exit` or hitting *Ctrl* + *d* to close the pane.

Windows are just like tabs

You can create a new window by keying in *Ctrl* + *b*, followed by *c*. You can now see that we have two panes at the bottom of the screen:

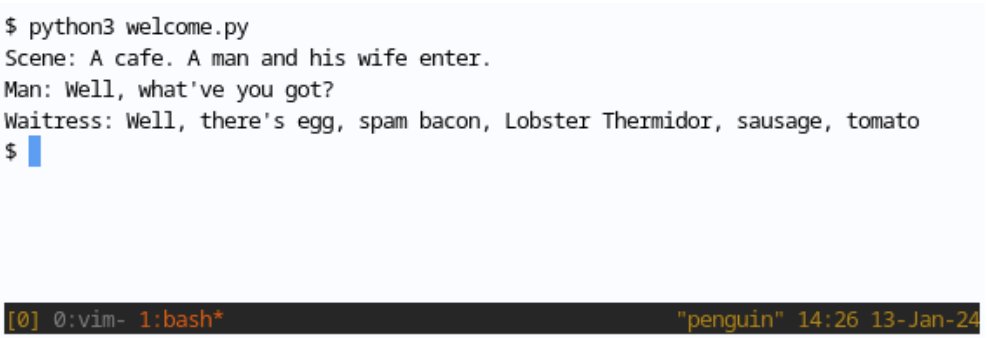

Figure 5.23 – **tmux** windows operate just like tabs in Vim.

Windows are automatically named based on what process is running in an active window within a pane. You can rename the current window by running *Ctrl* + *b*, followed by ,:

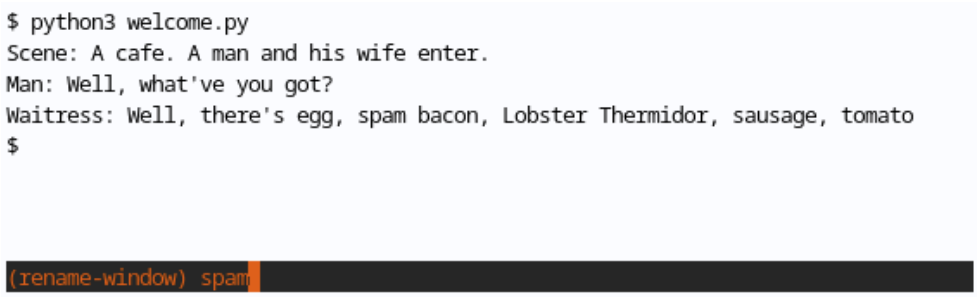

Figure 5.24 – You can rename a **tmux** window by running *Ctrl* + *b*, followed by a comma (,).

You can navigate the windows by pressing *Ctrl* + *b,* followed by *n*, to go forward and *Ctrl* + *b*, followed by *p*, to go backward.

Sessions are invaluable

If you SSH into a machine to work, tmux is an essential tool you can use. tmux allows you to create long-lasting sessions that outlive a single SSH connection.

If you're in a tmux session, you can detach from it by pressing *Ctrl* + *b*, followed by *d*. You will be sent back into your shell with the following message:

```
[detached (from session 0)]
```

The session will be alive until your machine is powered off. To list tmux sessions, execute the following:

```
$ tmux list-sessions
0: 2 windows (created Sat Jan 13 14:09:53 2024)
```

As you can see, we currently have one session available. Let's open the session, or—in tmux terms—attach to it:

```
$ tmux attach -t 0
```

Running `tmux` without any arguments always creates a new session.

You can have as many sessions as you want, if you prefer to separate work on different projects using different sessions! It's often helpful to divide projects or different tasks into sessions. You can navigate sessions from within tmux using *Ctrl* + *b*, followed by *(* or *)*.

You can also name your sessions, either when invoking tmux (`tmux new -s name`) or from within tmux (*Ctrl* + *b* and *$*).

tmux and Vim splits

Developers often use tmux panes and Vim windows to complement each other. You can have Vim open in different tmux panes as a way to isolate Vim instances from each other (and therefore group buffers). Normally, I have a few tmux panes open, with Vim in one of them (with its own splits as needed) and shell running in the rest. Everyone treats their panes and windows differently, and you may want to experiment.

tmux and Vim use different key bindings to navigate through windows (or, in tmux terminology, panes). This is rather confusing, and there are solutions in place to fix it! The easiest is to use the vim-tmux-navigator plugin, which is available from `https://github.com/christoomey/vim-tmux-navigator`. vim-tmux-navigator adds support for consistent navigation between Vim windows and tmux panes using the *Ctrl* + *h*, *Ctrl* + *j*, *Ctrl* + *k*, and *Ctrl* + *l* keys.

> **Tip**
> In order to use **vim-tmux-navigator**, your tmux needs to be version 1.8 or higher. You can check your tmux version by running the following:
>
> ```
> $ tmux -V
> ```
>
> See the previous *tmux* section for tips on how to install a newer version of tmux.

Installation instructions

If you're using vim-plug, you can install vim-tmux-navigator by adding the following line to your `.vimrc` file:

```
Plug 'christoomey/vim-tmux-navigator'
```

Don't forget to run `:w | source $MYVIMRC | PlugInstall` to install the plugin.

Once you have the plugin installed, you'll have to add the following bindings to your `~/.tmux.conf` file (this snippet is available from `https://github.com/christoomey/vim-tmux-navigator`):

```
# Smart pane switching with awareness of Vim splits.
# See: https://github.com/christoomey/vim-tmux-navigator
is_vim="ps -o state= -o comm= -t '#{pane_tty}' \
| grep -iqE '^[^TXZ ]+ +(\\S+\\/)?g?(view|n?vim?x?)(diff)?$'"
bind-key -n C-h if-shell "$is_vim" "send-keys C-h" "select-pane -L"
bind-key -n C-j if-shell "$is_vim" "send-keys C-j" "select-pane -D"
bind-key -n C-k if-shell "$is_vim" "send-keys C-k" "select-pane -U"
bind-key -n C-l if-shell "$is_vim" "send-keys C-l" "select-pane -R"
bind-key -T copy-mode-vi C-h select-pane -L
bind-key -T copy-mode-vi C-j select-pane -D
bind-key -T copy-mode-vi C-k select-pane -U
bind-key -T copy-mode-vi C-l select-pane -R
```

tmux has a plugin manager too!

If you're an advanced tmux user (I'm not), or don't mind digging around (I do), you may want to use TPM (Tmux Plugin Manager) instead of pasting the snippet into your `.tmux.conf` file. Add the following lines to `.tmux.conf` for TPM to configure the plugin for you:

```
set -g @plugin 'christoomey/vim-tmux-navigator'
run '~/.tmux/plugins/tpm/tpm'
```

You can learn more about TPM (and how to install it) from `https://github.com/tmux-plugins/tpm`.

Screen

Screen is tmux's spiritual predecessor, but it's still used by many today. Screen is not as extensible as tmux, and in fact, Vim doesn't play that nicely with Screen out of the box. However, if you're used to Screen and don't want to change your existing workflow, there are a few tweaks you can make to your setup to make the two get along a little nicer.

The *Esc* key doesn't register correctly in Vim when running through Screen. You might want to add the following to your `~/.screenrc` file to fix the *Esc* key behavior:

```
# Wait no more than 5 milliseconds when detecting an input
# sequence, fixes Esc behavior in Vim.
maptimeout 5
```

Screen also sets the `$TERM` variable to `screen`, which Vim does not recognize. Update `.screenrc` to include the following:

```
# Set $TERM to a value Vim recognizes.
term screen-256color
```

There are more minor inconveniences when using Vim and Screen together, such as the *Home* and *End* keys not registering, for example. Vim Wikia has a great in-depth entry on getting Vim and Screen to play along nicely at `http://vim.wikia.com/wiki/GNU_Screen_integration`.

Terminal mode

Historically, you could run shell commands from Vim by using `:!`, followed by a shell command. For example, we could execute our Python program as follows:

```
:!python3 welcome.py
```

Vim will pause, and you'll see the output in the Terminal:

```
~/Mastering-Vim-Second-Edition/Chapter05/spam$ vim welcome.py

Scene: A cafe. A man and his wife enter.
Man: Well, what've you got?
Waitress: Well, there's egg, spam bacon, Lobster Thermidor, sausage, tomato

Press ENTER or type command to continue
```

Figure 5.25 – You can run shell commands directly from Vim with `:!<command>`

Things have got better since then.

Starting in version 8.1, Vim introduced terminal mode. Terminal mode is effectively a Terminal emulator running within your Vim session. Unlike with tmux, terminal mode plays with Vim out of the box. It's great for running long-running commands while you continue to work in Vim.

Terminal mode can be invoked by executing the following:

```
:term
```

This opens a horizontal split with your default shell running:

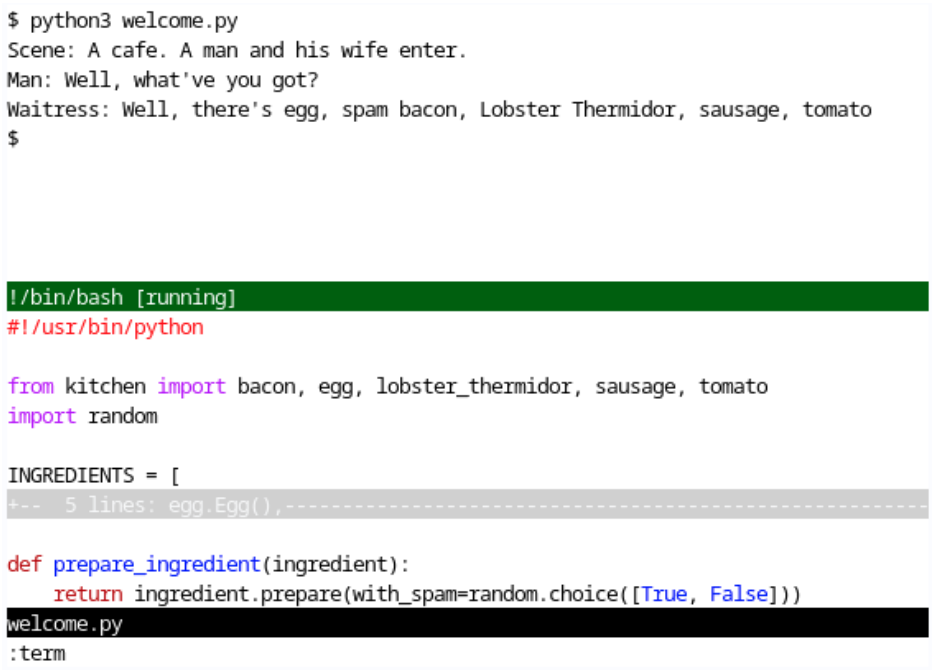

Figure 5.26 - `:term` opens a terminal split within Vim. It's convenient.

The Terminal window is treated like any other window and can be resized and moved as usual (see the *Windows* section in *Chapter 10*, *Neovim*). The window runs in a terminal-job mode, something akin to an insert mode. But there are a few specific key bindings:

- *Ctrl* + *w*, followed by *N*, enters a terminal-normal mode, which behaves just like a normal mode. Operations that take you back to insert mode (such as *i* or *a*) will take you back to a terminal-job mode.
- *Ctrl* + *w*, ", followed by a register, will paste the contents of a register into a terminal. For example, to paste something you yanked with *yw*, you can execute *Ctrl* + *w*, " to paste from the default register.
- *Ctrl* + *c* sends *Ctrl* + *c* to the running command in a Terminal.

The best feature of terminal mode is that you can invoke it with a specific command, and get full access to the output. Try running the following from within Vim:

```
:term python3 welcome.py
```

This will execute a command and, once it is done running, open a buffer with the resultant output:

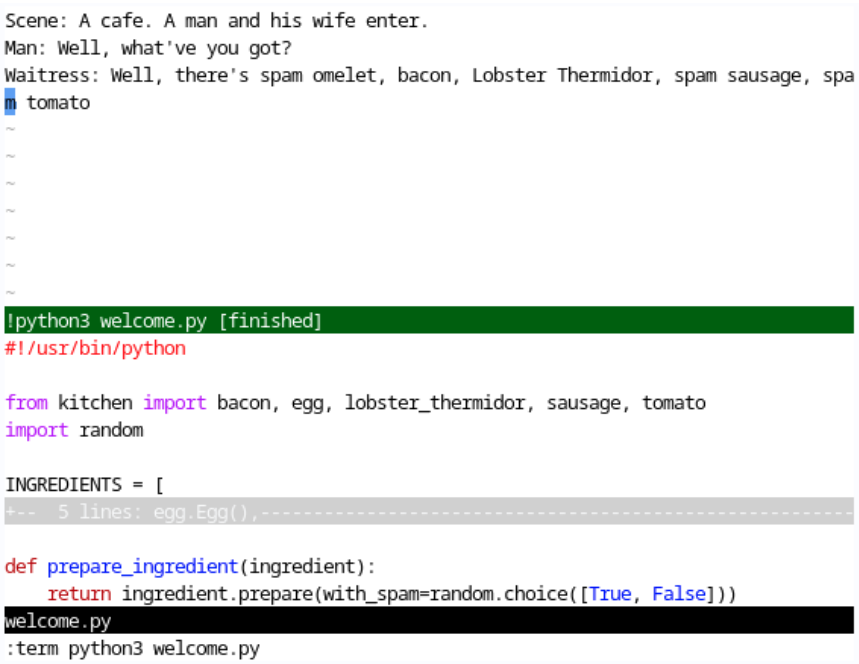

Figure 5.27 – Running `:term <command>` will execute the command and paste the results into a Vim buffer.

If you so desire, you can open a Terminal in a vertical split by running `:vert term`.

Making navigation easier

If you use the *Ctrl + hjkl* shortcuts from *Chapter 2*, *Advanced Editing and Navigation*, to navigate your Vim windows, you may want to add a set of bindings to your `.vimrc` file to work with terminal mode:

```
tnoremap <c-j> <c-w><c-j>
tnoremap <c-k> <c-w><c-k>
tnoremap <c-l> <c-w><c-l>
tnoremap <c-h> <c-w><c-h>
```

For best results, you can combine Vim terminal mode with tmux: tmux can manage your sessions for you (for when you need to switch focus between tasks), while terminal mode can manage windows. For instance, you could use Vim terminal windows to organize work on your project and tmux windows (or tabs in Vim terminology) to switch focus and work on different tasks.

Building and testing

As you work on your code, you will have to compile (in compiled languages, which does not apply to Python) it and run accompanying tests.

Vim supports populating build and test failures through quickfix and location lists, which we will cover in this section.

Quickfix list

You've already had a brush with a quickfix window in *Chapter 2*, *Advanced Editing and Navigation*, but let's dig a bit deeper into it.

Vim has an additional mode that makes jumping to certain parts of files easier. Some Vim commands use it to navigate between positions in files, such as jumping to compile errors for `:make` or search terms for `:grep` or `:vimgrep`. Plugins such as linters (syntax checking) or test runners use the quickfix list as well.

Let's try using a quickfix list by running a `:grep` command to search for the `ingredient` keyword recursively (`-r`) in every Python file (`--include="*.py"`), starting in the current directory (`.`):

```
:grep -r --include="*.py" ingredient .
```

This will open the first match in a current window. To open a quickfix window and see all of the matches, execute the following:

```
:copen
```

You can see the results in a horizontal split now:

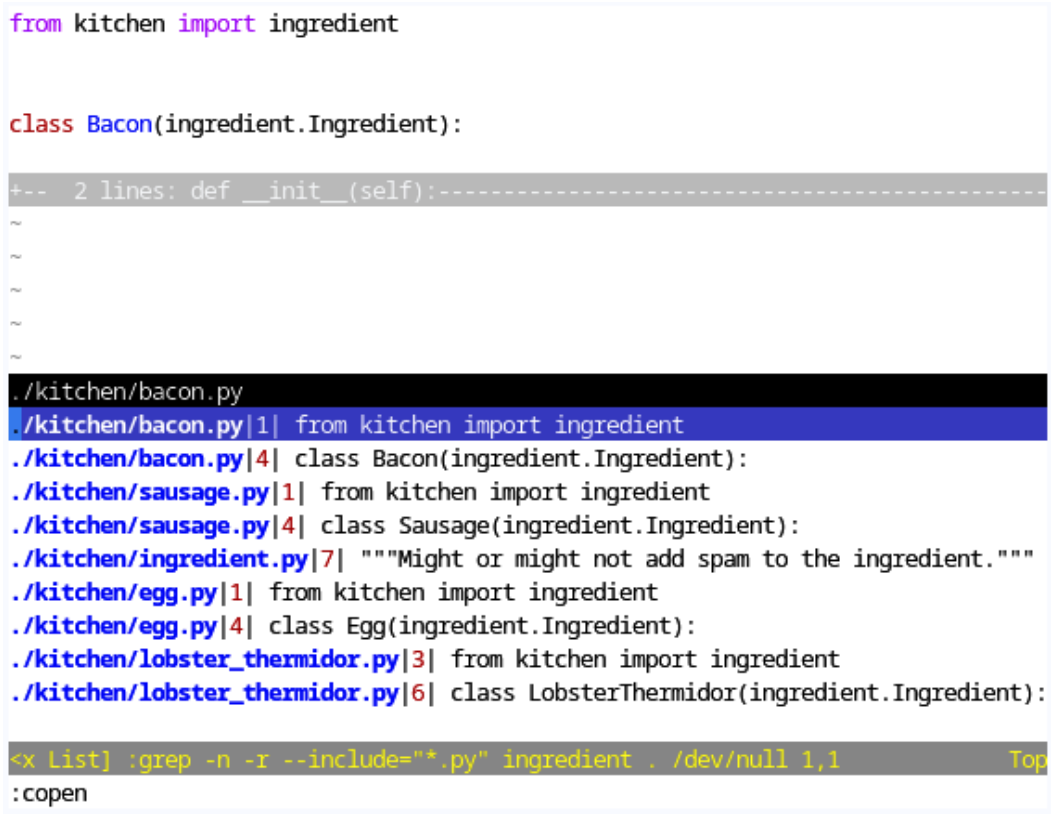

Figure 5.28 – Some Vim commands and plugins make use of a quickfix window, which can be opened with `:copen`.

You can navigate the quickfix window as usual with the *k* and *j* keys to move up and down, *Ctrl* + *f* and *Ctrl* + *b* to scroll by pages, and / and *?* to search forward and backward. *Enter* will open a file with a match in the buffer you were searching from. It will also place your cursor in the desired position.

For example, if you wanted to open a match in a file, `kitchen/egg.py`, you can navigate to the desired line by running `/egg`, followed by *n* (next) until the cursor is at the right line, and pressing *Enter*. The file will open in the original window with the cursor located where the match is:

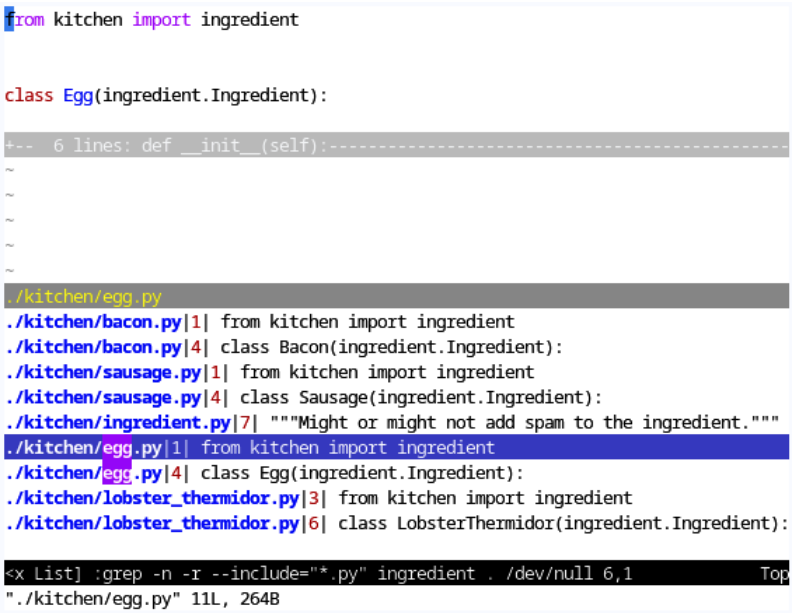

Figure 5.29 – The quickfix window allows you to navigate through the results of the `:grep` command.

You can close the quickfix list with `:cclose` (or `:bd` to delete the quickfix buffer if it's in an active window).

You can also navigate the quickfix list without opening the quickfix window:

- `:cnext` (or `:cn`) navigates to the next entry in the quickfix list
- `:cprevious` (or `:cp`, or `:cN`) navigates to the previous entry in the list

Lastly, you can choose to only open the quickfix window if errors (such as compile errors) are found: `:cwindow` (or `:cw`) will toggle the quickfix window only if errors are present.

Location list

In addition to a quickfix list, Vim also has a location list. It behaves just like a quickfix list, except that it stays local to the current window. While you can have only one quickfix list in a single Vim session, you can have as many location lists as you want.

To populate a location list, you can prefix most quickfix-operating commands with the letter `l` (such as `:lgrep` or `:lmake`).

Shortcuts also replace the `:c` prefix with the `:l` prefix:

- `:lopen` opens the location window
- `:lclose` closes the window
- `:lnext` navigates to the next item in a location list
- `:lprevious` navigates to the previous item in a location list
- `:lwindow` toggles the quickfix window only if the errors were present

In general, you will use a quickfix list when the results need to be accessed in multiple windows, while a location list is great for capturing output relevant to a single window.

Building code

Building doesn't necessarily apply to Python (since there isn't much compiling going on), but it's definitely worth going over to understand how Vim deals with executing code.

Vim provides a `:make` command, which wraps around the Unix make utility. In case you're not familiar, Make is a build management solution as old as time (and if it ain't broke...) that allows you to recompile parts of a bigger program (or all of it) as needed.

Some relevant options you'd want to be aware of are as follows:

- `:compiler` lets you specify a different compiler plugin, which also modifies the expected format output for the compiler
- In particular, `:set errorformat` defines a set of recognized error formats
- `:set makeprg` sets what program to execute when running `:make`

> **Don't forget about :help**
> Want to learn more about one of these options? Don't forget that you can run `:help <anything>` to look up an entry in the Vim manual.

The two can be used in conjunction to work with any compiler. For example, if you wanted to compile a C file you're working on, you could invoke `gcc` (the standard-issue C compiler) by running the following:

```
:compiler gcc
:make
```

What makes `:make` important is that it allows Vim users to implement syntax checkers, test runners, or just about anything else that spits out references to lines as a compiler plugin, giving us access to quickfix or location windows!

Terminal mode, introduced in Vim 8.1, is also a solid candidate for long-running builds, as `:term make` will call make asynchronously while you continue working on your code. See the *Terminal mode* section for more about terminal mode.

Testing code

Test output happens to be a lot less uniform than compile errors, so your best bet here is using test-runner-specific plugins you can find online. There are as many plugins as there are test runners, if not more.

In addition, terminal mode, added in Vim 8.1, provides a good way to run tests while continuing to work on your code.

Plugin spotlight – vim-test

This is the most popular test runner, as it provides a set of compilers (as well as handy mappings) for plugging into a lot of test runners. For Python, vim-test supports djangotest, django-nose, nose, nose2, pytest, and PyUnit. It's available from `https://github.com/janko-m/vim-test`. You'll have to make sure you have the desired test runner already installed before using vim-test.

> **Installing plugins**
> If you're using vim-plug, you can install vim-test by adding `Plug 'janko-m/vim-test'` to your `.vimrc` file and running `:w | source $MYVIMRC | PlugInstall`.

vim-test supports the following commands:

- `:TestNearest` runs the test nearest to the cursor
- `:TestFile` runs the tests in the current file
- `:TestSuite` runs the entire test suite
- `:TestLast` runs the last test

vim-test also allows you to specify test strategy, as in what method to use for running tests. Strategies such as make, neomake, MakeGreen, and dispatch (or `dispatch_background`) populate a quickfix window, which is exactly what you'd be looking for in a plugin like this.

For example, if you wanted to run your tests through vim-dispatch (to run a test in a different Terminal window, for instance), you would add the following to your `.vimrc` file:

```
let test#strategy = "dispatch"
```

You can visit `https://github.com/janko-m/vim-test` for more information about vim-test.

Syntax checking code with linters

Syntax checking (also known as linting) has essentially become a staple in any multi-person software project. There are many linter programs available online, which support different languages and styles.

Python code has it easier than many languages out there, as it tends to adhere to a single standard—PEP8 (`https://www.python.org/dev/peps/pep-0008`). The most common linters that make sure the code adheres to PEP8 are Pylint, Flake8, and autopep8.

Before proceeding, make sure one of these (the following examples work with Pylint) is installed on your machine, as Vim merely calls external linters.

> **Installing pylint**
> If you're on a Debian-flavored distribution, you can run `sudo apt install pylint3` to install Pylint for Python3.

Using linters with Vim

A lot of common linters have associated plugins, which you can use to avoid dealing with the intricacies of each linter. However, if you have to support a custom linter, Vim lets you populate a quickfix list however you want.

You can leverage Vim's `:make` command, which populates a quickfix list. By default, it runs the Unix `make` command (no surprise there), but you can override that by setting the `makeprg` variable.

Quickfix expects `:make` output to be in a particular format, and you can try to get a linter to output in a desired format. This is error-prone and has possible compatibility issues (if the underlying linter changes).

Add the following to your `.vimrc` file to override the `:make` behavior when only working on Python files:

```
autocmd filetype python setlocal makeprg=python3\ -m\ pylint\
--reports=n\ --msg-template=\"{path}:{line}:\ {msg_id}\ {symbol},\
{obj}\ {msg}\"\ %:p
autocmd filetype python setlocal errorformat=%f:%l:\ %m
```

Now, if you run `:make | copen` while in a Python file, you'll see a populated quickfix list:

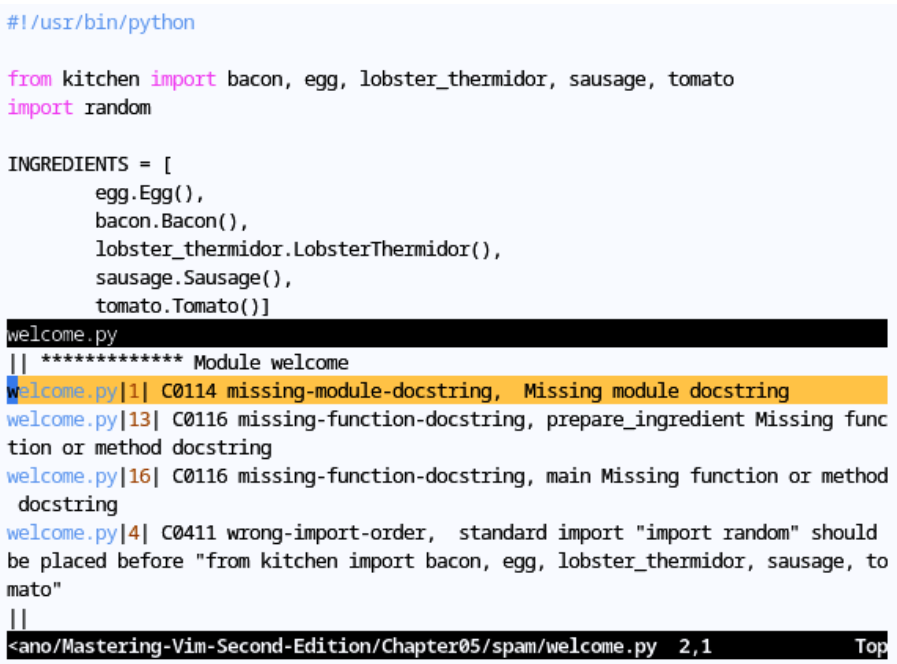

Figure 5.30 – The output of `:make | copen`: a quickfix list of Python lint errors.

Disabling linter warnings

If you're not accustomed to using linters, you might be wondering how to silence warnings you don't care for. For Pylint, it's done by adding a statement such as `disable-invalid-name,missing-docstring` to `~/.pylintrc`, or by commenting `# pylint: disable=invalid-name` on the offending line. Each linter has its own syntax for silencing warnings.

Plugin spotlight – ALE

Asynchronous Lint Engine (**ALE**) is a plugin that provides linting. Its primary selling point is that ALE displays lint errors as you type, and it runs the linters asynchronously. ALE is available from GitHub at `https://github.com/dense-analysis/ale`.

Installing plugins

If you're using **vim-plug**, you can install ALE by adding `Plug 'dense-analysis/ale'` to your `.vimrc` file and running `:w | source $MYVIMRC | PlugInstall`. ALE requires Vim 8+ or Neovim for asynchronous calls to work.

It's ready to be used out of the box. Here's a screenshot of a file with ALE enabled (I've opened the location window using `:lopen`):

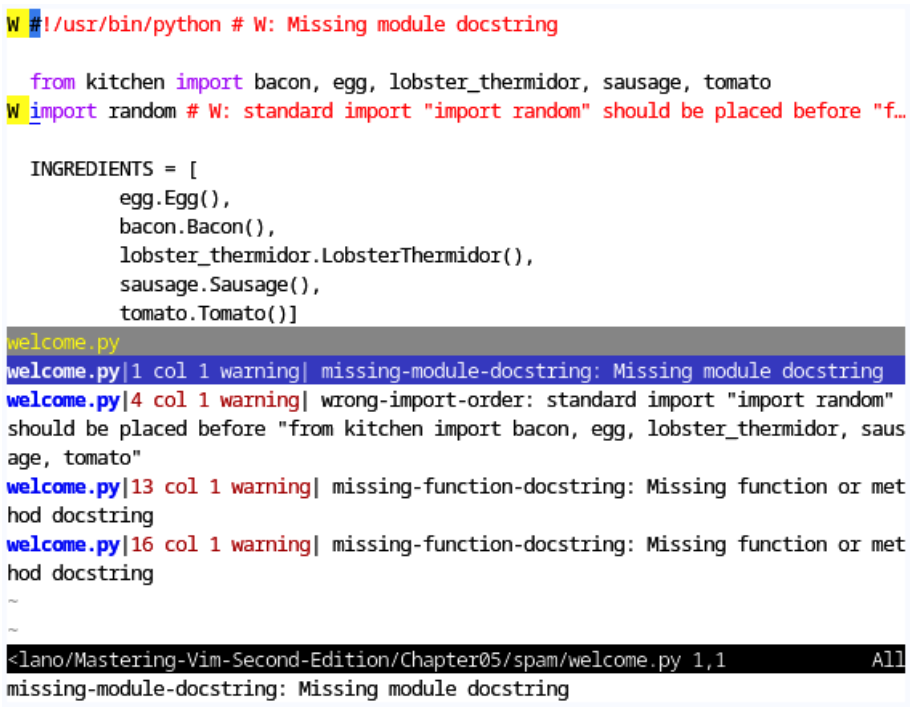

Figure 5.31 – The **ALE** plugin surfacing linter warnings. I should certainly be ashamed of my code.

You can see the line with an error highlighted with `>>`, and the status line displays the relevant lint message at the bottom.

You can toggle ALE on and off by running `:ALEToggle` if you don't like to be nagged by it.

ALE is a lot more than just a linter though, and is a full-blown language server protocol client: it supports autocompletion, traveling to definitions, and so on. It's not as established and popular as, say, YouCompleteMe (see the *Autocomplete* section in *Chapter 4*, *Understanding the Text*)—but it has a loyal fan base and has been growing rapidly.

For reference, you can jump to definitions by running `:ALEGoToDefinition` and looking for references using `:ALEFindReferences`. In order to enable autocomplete, you'll need the following line in your `.vimrc` file:

```
let g:ale_completion_enabled = 1
```

You can learn more about ALE and decide whether it's a tool worth investing your time in at `https://github.com/dense-analysis/ale`.

Summary

In this chapter, you learned (or got a refresher on) how to use Git, including a quick brush-up on its core concepts, setting up and cloning existing projects, and a rundown of the most frequent commands. You learned about vim-fugitive, a Vim plugin that makes Git a lot more interactive from inside Vim.

We covered vimdiff, a separate tool packaged with Vim, made for comparing files and moving changes between files. We learned how to compare and move changes between multiple files. Furthermore, we got some practice at resolving nasty Git merge conflicts, which will hopefully make them less intimidating.

This chapter covered multiple ways of running shell commands when working with Vim, be it through tmux, screen, or Vim terminal mode.

We also learned about (global) quickfix and (local) location lists, which can be used to store pointers to certain lines in files. We combined those with the output of the `:grep` and `:make` commands to get some easy-to-navigate results! We learned how `:make` works to call an external compiler, and we covered the vim-dispatch plugin to expand the `:make` functionality and the vim-test plugin to make running tests smoother.

Lastly, we covered a set of solutions for running syntax checkers in Vim, including building our own solution for Pylint. We also looked at ALE, an asynchronous linter.

In the next chapter, we will cover refactoring operations using Vim regular expressions and macros.

6

Refactoring Code with Regex and Macros

Vim's power extends beyond its extensive editing functionality. Macros allow you to record a series of keystrokes and replay them later, automating repetitive tasks. This chapter delves into the world of Vim macros and their application in refactoring code. In this chapter we'll explore how to create macros to streamline common refactoring operations, saving you time and effort while ensuring consistency in your codebase.

We will cover the following topics:

- Using search or replace functionality with `:substitute`
- Using regular expressions to make searches and substitutions smarter
- Using arglist to perform operations on multiple files
- Examples of refactoring operations, such as renaming methods and reordering arguments
- Macros, which let you record and replay keystrokes

Technical requirements

This chapter works with multiple code samples, which can be found on GitHub at `https://github.com/PacktPublishing/Mastering-Vim-Second-Edition/tree/main/Chapter06`.

You can work with that repository or, if you're feeling more comfortable, use your own project throughout this chapter.

Search or replace with regular expressions

Regular expressions (or regexes) are wonderful, and you should know how to use them. Vim, as is custom among regex implementations, has its own flavor of regex. However, once you learn one, you'll be comfortable with all of them.

First, let's talk about the regular (that is, the normal) search and replace command.

Search and replace

Vim supports search and replace through the `:substitute` command, most often abbreviated to `:s`. By default, `:s` will replace one substring with another in a current line. It has the following format:

```
:s/<find-this>/<replace-with-this>/<flags>
```

The flags are optional, and you shouldn't worry about them for now. To try it, open `welcome.py`, navigate to the line containing `egg` (for example, with `/egg`), and execute the following:

> **Reminder**
>
> `/` followed by a substring allows you to search for a substring and will move the cursor to the line of the first match.

```
:s/egg/omelette
```

As you can see in the following screenshot, this replaces the first occurrence of `egg` in the current line (line 3) with `omelette`:

```
#!/usr/bin/python

from kitchen import bacon, omelette, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

:s/egg/omelette
```

Figure 6.1 – `egg` was replaced with `omelette` in the selected line.

Now, let's look at the flags you can pass to the substitute command:

- `g`—global replace: replace every occurrence of the pattern in the line, not just the first one
- `c` —confirm each substitution: prompt the user before replacing the text
- `e` —do not show errors if no matches are found
- `i` —ignore case: make the search case-insensitive
- `I` —make the search case-sensitive

You can mix and match these (except for `i` and `I`) as you see fit. For example, running `:s/egg/omelette/gi` will turn the string `egg.Egg()` into `omelette.omelette()`.

> **Preserving case in substitutions**
>
> Interested in how one would preserve case and replace `egg.Egg()` with `omelette.Omelette()`? It's non-trivial to do with native Vim regex, but I recommend looking up the plugin called **abolish.vim** (from `https://github.com/tpope/vim-abolish.git`), which does just that!

`:substitute` can be prefixed by a range, which tells it what to operate on. The most common range used with `:substitute` is `%`, which makes `:s` operate on the current file.

For instance, if we wanted to replace each instance of `ingredient` in a file with `food`, we would run the following:

```
:%s/ingredient/food/g
```

If you try it on `welcome.py`, you'll see, as in the screenshot, that every instance of `ingredient` was replaced with `food`:

```
#!/usr/bin/python

from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_food(food):
    has_spam = random.choice([True,  False])
    return food.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for food in INGREDIENTS:
        menu.append(prepare_food(food))
    print('Waitress: Well, there\'s', ', '.join(menu))


if __name__ == '__main__':
    main()
~
6 substitutions on 4 lines
```

Figure 6.2 – Adding `%` before the `:s` command applies substitutions to the whole file.

The `:substitute` command conveniently tells us how many matches were replaced in the status line at the bottom of the screen.

It seems as if we just completed a very simple case of refactoring!

`:substitute` supports more ranges. Here are some common ones:

- numbers—a line number
- `$`—the last line in the file
- `%`—a whole file (this is one of the most used ones)
- `/search-pattern/`—lets you find a line to operate on
- `?backwards-search-pattern?`—does the same thing as the previous flag, but searches backward

> **Tip**
> Prefixing ranges like this works with other commands in Vim! See `:help cmdline-ranges` for more info.

Furthermore, you can combine the ranges with a `,` operator. For example, `20,$` will let you search from *line 20* until the end of the file.

To demonstrate on a somewhat derived example, the following command will search for and replace every instance of `ingredient` with `food` from *line 8*, up to and including the line where it encounters `main`:

```
:8,/main/s/ingredient/food/g
```

> **For a discerning reader**
> You may have noticed competing British and American English spellings of the word "`omelette/omelet`". If you're reading this comment, I've successfully convinced the editor that this is an Easter egg and is most certainly not my inability to spell.

As you can see in the following screenshot, three instances of `ingredient` were replaced on *lines 8* and *10*, but not on *lines 16* or *17* (I've enabled line number display by running `:set nu`):

Figure 6.3 – Command-line ranges allow you to control where to apply the substitutions.

You can also select a range in a visual mode, and run `:s` without any explicit ranges to operate on a selected text. See `:help cmdline-ranges` for more information on ranges.

> **Tip**
>
> If you find yourself working with Linux file paths (or anything with `/` in them), you can escape them by prefixing with a backslash (`\`) or changing the separator. For example, `:s+path/to/dir+path/to/other/dir+gc` is (with a separator changed to `+`) equivalent to `:s/path\/to\/dir/path\/to\/other\/dir/gc`.

Most often, you will find yourself replacing all occurrences in the whole file by running the following:

```
:%s/find-this/replace-with-this/g
```

When replacing text, you may want to only search for the whole word. You can use `\<` and `\>` for this purpose. For example, given the following file, we can search for `/ingredient` (`:set`

`hlsearch` is enabled to highlight all results), but we also get results we're not exactly interested in, such as `prepare_ingredient`:

Figure 6.4 – By default, search and substitutions will return partial matches like `prepare_ingredient` when searching for `ingredient`.

However, if we search for `/\<ingredient\>`, we'll be able to match whole words only, without falsely detecting instances of `prepare_ingredient`, as follows:

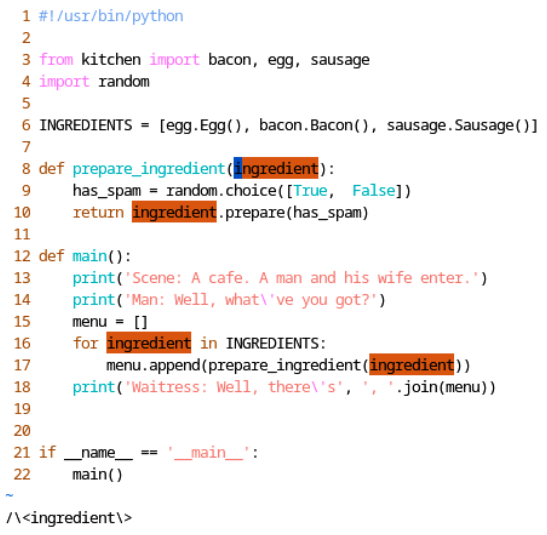

Figure 6.5 – Surrounding the search term with \< and \> will return complete matches only.

Operations across files using arglist

If you give more than one file name when starting Vim, this list is remembered as an argument list. **Arglist** allows you to perform the same operation on multiple files, without having to have them preloaded in buffers first.

Arglist provides a few commands, as follows:

- `:arg` defines the arglist.
- `:argdo` allows you to execute a command on all the files in the arglist.
- `:args` displays the list of files in the arglist.

For example, if we wanted to replace all instances of `ingredient` in every Python file (recursively), we would do the following:

```
:arg **/*.py
:argdo %s/\<ingredient\>/food/ge | update
```

> **Extra credit**
>
> Since **arglist** is populated when you open Vim, you could also open the files in the command line (`vim **/*.py`), and then run `argdo`. For extra credit, this can be done as a single command from the command line: `vim **/*.py -c ":argdo :%s/<ingredient>/food/ge|update"`. Here, the `-c` option allows you to execute a command after opening Vim.

This command works as follows:

- `:arg <pattern>` adds a set of files matching a pattern to the arglist (each argument in arglist also has a corresponding buffer).
- `**/*.py` is a wildcard for every `.py` file, recursively starting with the current directory.
- `:argdo` executes a command on every item in the argument list.
- `%s/\<ingredient\>/food/ge` replaces every occurrence of `ingredient` with `food`, in every file, without raising errors if the matches are not found.

> **Tip**
>
> As mentioned above, `\<` and `\>` around `ingredient` tell Vim to only replace a whole word, so we won't be replacing occurrences like `prepare_ingredient`.

`:update` is equivalent to `:write`, but it only saves the file if the buffer has been modified.

You need to use `:update` in arglist commands, because Vim doesn't like it when you switch buffers without saving their contents. An alternative would be to use `:set hidden` to silence these warnings and save all files at the end by running `:wa`.

Give it a shot, and you'll see that every occurrence of a word has been replaced (you can check by running `git status` or `git diff` if you have a repository checked into Git). You can also view the contents of the arglist by running the following without any arguments:

```
:args
```

Technically, you can also use `:bufdo` to perform an operation on every open buffer (since arglist entries are reflected in the buffer list). However, I would advise against it, since there is a risk of running a command on buffers you unintentionally had open before populating the argument list.

Regex basics

Regular expressions work in substitution commands, as well as in search. Regex introduces special patterns that can be used to match a set of characters; for example, see the following:

- `\(c\|p\)arrot` matches both `carrot` and `parrot`—the `\(c\|p\)` denotes either `c` or `p`.
- `\warrot\?` matches `carrot`, `parrot`, and even `farro`—the `\w` signifies any word character, and the `t\?` means that the `t` is optional.
- `pa.\+ot` matches `parrot`, `patriot`, or even `pa123ot`—the `.\+` denotes one or more of any character.

> **Note**
> Learning regular expressions takes some time and effort, and I only touch on a couple of Vim-specific examples in this chapter. But that's the benefit of being an author of this book: Vim or not—I think regex is useful, important, and will be taught to you. Enjoy!

> **Tip**
> If you're familiar with other variations of regex, then you'll notice that unlike in many other regex flavors, most special characters need to be escaped with `\` to work (the default mode for most characters is non-regex, with a few exceptions such as `.` or `*`). This behavior can be reversed by using magic mode, as we will cover below.

Special regex characters

Let's dig deeper into regex:

Symbol	Meaning
`.`	Any character, except for end of the line
`^`	The beginning of the line
`$`	The end of the line
`\_.`	Any character, including end of the line
`\<`	The beginning of a word
`\>`	The end of a word

> **Tip**
> You can see the full list of these using `:help ordinary-atom`.

There are also what Vim calls character classes:

Symbol	Meaning
`\s`	Whitespace (Tab and Space)
`\d`	A digit
`\w`	A word character (digits, numbers, or underscores)
`\l`	A lowercase character
`\u`	An uppercase character
`\a`	An alphabetic character

These classes have the opposite effect when capitalized; for example, `\D` matches all non-digit characters, whereas `\L` matches everything but lowercase letters (note that this is different from just matching uppercase letters).

> **Tip**
> You can see the full list by checking out `:help character-classes`.

You can also specify a set of characters explicitly, using square brackets (`[]`). For instance, `[A-Z0-9]` will match all uppercase characters and all digits, while `[,4abc]` will only match commas, the number `4`, and letters `a`, `b`, and `c`.

For sequences (such as numbers or letters of the alphabet), you can use a hyphen (`-`) to represent a range. For instance, `[0-7]` will include numbers from `0` to `7`, and `[a-z]` will include all lowercase letters from `a` to `z`.

Here's one more example, including letters, numbers, and underscores: `[0-9A-Za-z_]`.

Finally, you can negate an entire range by prefixing it with a caret (`^`). For instance, if you wanted to match all non-alphanumeric characters, you would put `[^0-9A-Za-z]`.

Alternation and grouping

Vim has a few more special operators:

Symbol	Meaning
`|`	alternation
`\(\)`	grouping

The alternation operator is used to signify *or*. For example, `carrot\|parrot` matches both `carrot` and `parrot`.

Grouping is used to put multiple characters in a group, which can serve two purposes. Firstly, you can combine operators with each other. For example, `\(c\|p\)arrot` is a nicer way to match both `carrot` and `parrot`.

Grouping can also be used to later refer to each section in parentheses. For example, if you wanted to turn the string `cat hunting mice` into `mice hunting cat`, you could use the following `:substitute` command:

```
:s/\(cat\) hunting \(mice\)/\2 hunting \1
```

Here, `\1`, `\2`, and so on refer to **capture groups** – the content within parentheses. `\1` contains `cat` and `\2` contains `mice`.

Grouping becomes relevant during refactoring, for example, when reordering arguments—but more on that later.

Quantifiers or multis

Each character (be it a literal or a special character) or a range of characters is followed by a quantifier, or a *multi* in Vim terms.

For example, `\w\+` will match one or more word characters, and `a\{2,4}` will match two to four `a` characters in succession (such as `aaa`, for example).

Here is a list of the most common quantifiers:

Symbol	Meaning
`*`	0 or more, greedy
`\+`	1 or more, greedy
`\{-}`	0 or more, non-greedy

Symbol	Meaning
`\? or \=`	0 or 1, greedy
`\{n,m}`	n to m, greedy
`\{-n,m}`	n to m, non-greedy

> **Tip**
> The full list of quantifiers is available through `:help multi`.

You may have encountered two new terms in the table given: greedy and non-greedy. **Greedy** search refers to trying to match as many characters as possible, while **non-greedy** search tries to match as few characters as possible.

For example, given a string `foo2bar2`, greedy regex `\w\+2` will match `foo2bar2` (as many characters as it can until encountering a final `2`), while non-greedy `\w\{-1,}2` will only match `foo2`.

More about magic

Escaping special characters with backslashes `\` is no trouble if you're only occasionally spicing up your searches and substitutions with regular expressions. If you want to write longer expressions without having to escape every special character, you could switch to the magic mode for that expression.

Magic mode determines how Vim parses regex-enabled strings (like those in search or substitute commands).

Vim has three magic modes: magic, no magic, and very magic.

Magic

This is the default mode. Most special characters need to be escaped, but some (such as `.` or `*`) don't have to be.

You can prefix your regex strings with `\m` (for example, `/\mfoo` or `:s/\mfoo/bar`) to explicitly set magic.

No magic

This mode is similar to magic mode, but every special character needs to be escaped with a backslash, `\`.

For example, in default magic mode, you'd search for a line containing any text with `/^.*$` (here, `^` is for the beginning of a line, `.*` searches for every character repeatedly, and `$` is for the end of a line). In no-magic mode, this pattern would translate to `/\M^\.\*$`.

You can explicitly set no-magic mode by prefixing your regex strings with `\M` (for example, `/\Mfoo` or `:s/\Mfoo/bar`). No magic can also be set in your `.vimrc` by adding `set nomagic`, but it's highly discouraged; by changing the way Vim treats regular expressions, you're more than likely to break several plugins you're using (as their creators will not have built them to work in no magic mode).

Very magic

Very magic mode treats every character apart from letters, numbers, and underscores as a special character.

You can set the very magic mode for a command by prefixing your regex strings with `\v` (for example, `/\vfoo` or `:s/\vfoo/bar`).

Very magic mode is often used when many special characters are to be used. For instance, we used the following example to replace `cat hunting mice` with `mice hunting cat`:

```
:s/\(cat\) hunting \(mice\)/\2 hunting \1
```

In very magic mode, this can be rewritten as follows:

```
:s/\v(cat) hunting (mice)/\2 hunting \1
```

Applying the knowledge in practice

Many tasks when refactoring code involve renaming or reordering things, and regular expressions are perfect tools for this.

Renaming a variable, a method, or a class

Oftentimes, we rename things when refactoring, and these changes need to be reflected throughout the codebase. However, simple search and replace often won't cut it, since you'll risk accidentally renaming unrelated things.

For example, let's try renaming our `Egg` class as `Omelette`. Since we need to carry this out in multiple files, we'll use arglist to load all the Python files into Vim buffers:

```
:arg **/*.py
```

Now, move your cursor over the class name you'd like to rename (`Egg`), and enter the following (here, `\<[Ctrl + r, Ctrl + w]\>` signifies pressing *Ctrl* + *r* followed by *Ctrl* + *w* and not typing in the square brackets):

```
:argdo %s/\<[Ctrl + r, Ctrl + w]\>/Omelette/gec | update
```

Once you run it, you'll be prompted for every match:

```
from kitchen import ingredient


class Egg(ingredient.Ingredient):

    def __init__(self):
        self.name = 'egg'

    def prepare(self, with_spam=True):
        """Becomes an omelet as you add spam!"""
        return 'spam omelet' if with_spam else self.name
replace with Omelette (y/n/a/q/l/^E/^Y)?
```

Figure 6.6 – You will be prompted for every match because you're using the `c` regexp flag.

Press *y* to approve each change, or *n* to reject it.

Here's what's going on here:

- `:argdo` runs the operation on every arglist entry (which we loaded with `:arg`)
- `%s/.../.../gec` substitutes every occurrence (`g`) throughout the whole file (`%`), without raising errors if no entries were found (`e`), and asking the user before making changes (`c`)
- `\<...\>` ensures we're looking for a whole word, and not just partial matches (otherwise we'll also rename another class like `Eggnog`, which we don't want to do)
- *Ctrl* + *r*, *Ctrl* + *w* is a shortcut to insert the word under the cursor in the current command (which would insert `Egg`)

This approach has the disadvantage of locking you into dialog windows, without you being able to look around the file first. If you'd like more control, another alternative would be to use `:vimgrep` to find the matches first:

```
:vimgrep /\<Egg\>/ **/*.py
```

You'll be able to look at matches and step through them with `:cn` or `:cp` (or open the quickfix window with `:copen` and navigate from there):

```
from kitchen import ingredient


class Egg(ingredient.Ingredient):

    def __init__(self):
        self.name = 'egg'

    def prepare(self, with_spam=True):
        """Becomes an omelet as you add spam!"""
        return 'spam omelet' if with_spam else self.name
(1 of 2): class Egg(ingredient.Ingredient):
```

Figure 6.7 – `:vimgrep` outputs its content into a quickfix window.

> **More commands**
> You can also use the `:cdo` or `:cfdo` command to iterate over all lines in the quickfix list.

In this particular example, you could replace the word using the usual change word command (`cw` followed by `Omelette` followed by *Esc*), and then replay the changes by pressing dot (`.`), or run a non-global `:substitute` command (`:s/\<Egg\>/Omelette`).

Regex can be a very powerful tool in your arsenal when combined with other Vim features. For example, you could delete all HTML tags in the document by running `:%s/<[^>]*>//g`, or remove single-line comments (starting with #) by running `:%s#//.*$##`. Your imagination is the limit!

Recording and playing macros

Macros are an extremely powerful tool that allows you to record and replay a set of actions. They're especially helpful for a certain type of manual task: something too laborious to perform by hand, but not complex enough to warrant a script or a program. There's a Goldilocks zone for when macros are the most useful, and you'll get a good feel for when that is.

Here's a typical example of when a macro would be helpful. An intern tried to author a program that pays homage to Monty Python's "spam" sketch: almost every meal in the cafe is made with spam! But the intern (of course this wasn't you!) wrote some no-good code. Just look at the blatant violation of the do-not-repeat-yourself principle:

```
#!/usr/bin/python

from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    if isinstance(ingredient, egg.Egg) and has_spam:
        return 'spammy eggs'
    if isinstance(ingredient, bacon.Bacon) and has_spam:
        return 'bacon and spam'
    if isinstance(ingredient, sausage.Sausage) and has_spam:
        return 'spam sausage'
    return ingredient.name

def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for ingredient in INGREDIENTS:
        menu.append(prepare_ingredient(ingredient))
    print('Waitress: Well, there\'s', ', '.join(menu))
```

Figure 6.8 – Some not-so-elegant code demonstrating a repetitive pattern.

The intern's confusion is understandable: there's no consistent naming scheme across dishes! We have spammy eggs, bacon and spam, and spam sausage! But we can help them write better code.

There are many ways to clean up the code, one of which is to move the custom spam-flavored ingredient names into each corresponding ingredient class. That'll make `prepare_ingredient` cleaner!

To do this, we'll probably want to expand `Ingredient` from `kitchen/ingredient.py` with a generic `prepare` method:

```
class Ingredient(object):

    def __init__(self, name):
        self.name = name
        self.custom_spam_name = None

    def prepare(self, with_spam=True):
        """Might or might not add spam to the ingredient."""
        if with_spam:
            return self.custom_spam_name or 'spam ' + self.name
        return self.name
~
"kitchen/ingredient.py" 11L, 331B written
```

Figure 6.9 – `Ingredient.prepare` uses a custom spam dish name if it exists, or defaults to a simple name like "spam egg".

Back in `welcome.py`, this will leave `prepare_ingredient` looking cleaner:

```
#!/usr/bin/python

from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    return ingredient.prepare(with_spam=has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
    print('Man: Well, what\'ve you got?')
    menu = []
    for ingredient in INGREDIENTS:
        menu.append(prepare_ingredient(ingredient))
    print('Waitress: Well, there\'s', ', '.join(menu))


if __name__ == '__main__':
    main()
~
~
"welcome.py" 22L, 571B written
```

Figure 6.10 – `prepare_ingredient` looking much cleaner by delegating responsibility to the `ingredient.Ingredient` class.

All that's left is to move existing names into `custom_spam_name` of each corresponding class (`egg.Egg`, `bacon.Bacon`, and `sausage.Sausage`). And that's where macros come in!

Let's go back to `welcome.py` written by our intern and place our cursor on the beginning of *line 10*.

```
1  #!/usr/bin/python
2
3  from kitchen import bacon, egg, sausage
4  import random
5
6  INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]
7
8  def prepare_ingredient(ingredient):
9      has_spam = random.choice([True,  False])
10     if isinstance(ingredient, egg.Egg) and has_spam:
11         return 'spam eggs'
12     if isinstance(ingredient, bacon.Bacon) and has_spam:
13         return 'bacon and spam'
14     if isinstance(ingredient, sausage.Sausage) and has_spam:
15         return 'spam sausage'
16     return ingredient.name
17
18 def main():
19     print('Scene: A cafe. A man and his wife enter.')
20     print('Man: Well, what\'ve you got?')
21     menu = []
22     for ingredient in INGREDIENTS:
23         menu.append(prepare_ingredient(ingredient))
24     print('Waitress: Well, there\'s', ', '.join(menu))
```

Figure 6.11 – `welcome.py`, note the cursor position on line 10.

Here's what we'll want to do:

1. Copy the contents of the return statement (`'spam eggs'`)
2. Navigate into the related class (`egg.Egg`)
3. Set the class property: `self.custom_spam_name = 'spam eggs'`

Macros will allow you to record and later replay every command you input (including movement), and because of that, we'll need to be precise and deliberate with our movements.

The following instructions do take up a couple of pages of the book, but it can be chalked up to overexplaining and screenshots. Once you get the gist, macros are quite simple to use.

Let's break down each step:

1. Copy the contents of the return statement (`'spam eggs'`)

   A. *qa* – Begin recording a macro into register `a`

```
    if isinstance(ingredient, egg.Egg) and has_spam:
        return 'spam eggs'
    if isinstance(ingredient, bacon.Bacon) and has_spam:
recording @a
```

   Note `recording @a` in the status line, which signifies that you're recording a macro into the register `a`.

   B. *j* – Move down to the line containing return `'spam eggs'`

```
    if isinstance(ingredient, egg.Egg) and has_spam:
        return 'spam eggs'
    if isinstance(ingredient, bacon.Bacon) and has_spam:
```

   C. _ – Move the cursor to the first character of the line

```
    if isinstance(ingredient, egg.Egg) and has_spam:
        return 'spam eggs'
    if isinstance(ingredient, bacon.Bacon) and has_spam:
```

   D. *w* – Move the cursor by one word

```
    if isinstance(ingredient, egg.Egg) and has_spam:
        return 'spam eggs'
    if isinstance(ingredient, bacon.Bacon) and has_spam:
```

   E. *"by$* – Yank the text until the end of the line (*y$*) into the register `b` (*"b*), as the register `a` is in use for the macro. Now register `b` will contain `'spam eggs'`.

```
    if isinstance(ingredient, egg.Egg) and has_spam:
        return 'spam eggs'
    if isinstance(ingredient, bacon.Bacon) and has_spam:
```

   F. You could technically yank into a default register, but it's easy to override the default register when editing – which is why I prefer to copy into a dedicated register.

2. Navigate into the related class (`egg.Egg`)

   A. *k* – Move one line up

```
    if isinstance(ingredient, egg.Egg) and has_spam:
        return 'spam eggs'
    if isinstance(ingredient, bacon.Bacon) and has_spam:
```

B. *f)* – Move to the closing parenthesis

```
if isinstance(ingredient, egg.Egg) and has_spam:
    return 'spam eggs'
if isinstance(ingredient, bacon.Bacon) and has_spam:
```

C. *b* – Move one word backward

```
if isinstance(ingredient, egg.Egg) and has_spam:
    return 'spam eggs'
if isinstance(ingredient, bacon.Bacon) and has_spam:
```

D. *Ctrl* + *]* – Jump to the class definition, which should take us to another file.

```
from kitchen import ingredient


class Egg(ingredient.Ingredient):

    def __init__(self):
        self.name = 'egg'
"kitchen/egg.py" 7L, 118B
```

3. Set the class property: `self.custom_spam_name = 'spam eggs'`

A. `/self.name` – Place the cursor on the line where `self.name` is defined

```
from kitchen import ingredient


class Egg(ingredient.Ingredient):

    def __init__(self):
        self.name = 'egg'
/self.name
```

B. *o* – Enter insert mode on the next line.

```
class Egg(ingredient.Ingredient):

    def __init__(self):
        self.name = 'egg'

-- INSERT --
```

C. Type `self.custom_spam_name = `, followed by the *Escape* key.

```
class Egg(ingredient.Ingredient):

    def __init__(self):
        self.name = 'egg'
        self.custom_spam_name =
```

D. *"bp* – Paste contents of the register `b`.

```
class Egg(ingredient.Ingredient):

    def __init__(self):
        self.name = 'egg'
        self.custom_spam_name = 'spam eggs'
```

E. *:w* – Save the file.

```
class Egg(ingredient.Ingredient):

    def __init__(self):
        self.name = 'egg'
        self.custom_spam_name = 'spam eggs'
"kitchen/egg.py" 8L, 162B written
```

4. Return to `welcome.py` and remove the offending lines.

A. *Ctrl* + ^ – Open the previous file.

```
def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    if isinstance(ingredient, egg.Egg) and has_spam:
        return 'spam eggs'
    if isinstance(ingredient, bacon.Bacon) and has_spam:
        return 'bacon and spam'
"welcome.py" 28L, 808B
```

Personally, I can never remember the *Ctrl* + ^ shortcut, but I do remember how to navigate Vim's navigation stack with *Ctrl* + *o* and *Ctrl* + *i*. So I just hit *Ctrl* + *o* until I'm placed in the previous file.

B. *2dd* – Delete two lines.

```
def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    if isinstance(ingredient, bacon.Bacon) and has_spam:
        return 'bacon and spam'
    if isinstance(ingredient, sausage.Sausage) and has_spam:
        return 'spam sausage'
"welcome.py" 28L, 808B
```

C. *q* – Stop recording the macro.

That's it. You now have this macro recorded in register `a`!

Let's try replaying the macro by hitting @, followed by *a*: *@a*. Behold your actions quickly being replayed in front of you.

> **Registers**
> We talk more about registers in the *Copying and pasting with registers* section of *Chapter 2, Advanced Editing and Navigation*.

> **Tip**
> A handy shortcut is @@. @@ replays the last macro you ran.

Repeating macros

Say you have a simple macro that prefixes `spam` to each dish name in a list:

```
dish_names = [
    'omelet',
    'sausage',
    'bacon'
]
~
~
~
~
~
~
```

Figure 6.12 – A list of dish names, to be prefixed by the everpresent "spam".

You'll record something like this:

- *qa* – Begin recording macro into the register `a`.
- */ ‘* – Find the next single quote (‘) prefixed by four spaces.
- *_* – Go to the beginning of the line (where text is present).
- *a* – Enter insert mode after the cursor.
- `spam ` – Type `spam` followed by a space.
- *Esc* – Exit to normal mode.
- *q* – Finish recording the macro.

You can repeat the macro multiple times by prefixing it with a number: *2@a*. However, if, for instance, you are searching as part of your macro (try rerunning the macro once more with *@a*), the search may wrap back to the beginning of the file and replay a macro on a portion of the file you've already modified:

```
dish_names = [
    'spam spam omelet',
    'spam sausage',
    'spam bacon'
]
~
~
~
~
~
~
```

Figure 6.13 – Unexpected spam in our spam omelet!

That's where working with macros can get messy. All macros do is record your actions and replay them back.

So, how can we make this macro not do this?

A macro stops executing if it encounters an error. If there are no patterns we're searching for below the cursor, Vim just looks for one above the cursor—without producing an error. So we just need to manually produce an error, to make sure the macro doesn't continue running when it doesn't have to.

```
:set nowrapscan
```

If you replay the macro, you'll now get an error:

```
dish_names = [
    'spam omelet',
    'spam sausage',
    'spam bacon'
]
~
~
~
~
~
~
E385: Search hit BOTTOM without match for:     '
```

Figure 6.14 – Sometimes it's useful to deliberately trigger an error.

Now you can safely execute this macro any number of times.

Due to errors like these, or if you're not confident about the matches, sometimes it's useful to carry out a separate search. It might make sense to search for an occurrence outside the macro, and then play the macro if you decide that the change is warranted. Then, you can run `n` to search for the next search occurrence, decide whether you'd like to make changes, and run the macro again with `@a` or `@@`.

Editing macros

Macros are stored in registers (the same ones used by yank and paste operations). You can view the contents of all your registers by executing `:reg`:

```
l  "5          self.custom_spam_name = 'bacon and spam'^J
l  "6          self.custom_spam_name = 'bacon and spam'^J
l  "7          self.custom_spam_name = 'bacon and spam'^J
l  "8          self.custom_spam_name = 'bacon and spam'^J
l  "9          self.custom_spam_name = 'bacon and spam'^J
c  "a   j_w"by$kf)<80><fd>ab^]/self.name^Moself.custom_name <80>kb<80>kb<80>k
c  "b   'spam sausage'
c  ":   q
c  "%   welcome.py
c  "#   kitchen/bacon.py
c  "/   "q
Press ENTER or type command to continue
```

Figure 6.15 – Output of :reg command, listing the content of all registers.

It's a bit messy, but close to the middle of the list you can see `"a`, the register containing our macro. You can also view the contents of, say, register `a` by executing `:echo @a` or `:reg a`.

In the preceding screenshot, many special characters are represented differently. For instance, `^[` signifies the *Esc* key, and `^M` is an *Enter* key.

In fact, macros are nothing but registers: the *q* command lets you add keystrokes to the register, while @ lets you replay the keystrokes from that register.

Since the macro is effectively a register, you can paste it using `p`. Open a new buffer with `:new`, and paste the contents of the register using `"ap`:

```
j_w"by$kf)<80><fd>ab^]/self.name^Moself.custom_name <80>kb<80>kb<80>kb<80>kb<80>
kbsanm<80>kb<80>kb<80>kbpam_name = ^["b^["bp:w^M^^2dd
~
~
~
~
~
~
~
~
~
```

Figure 6.16 – Content of the macros `a` that we recorded earlier.

Now you can edit your macro without having to retype the whole thing.

When you're finished editing, copy it back into the register: *_"ay$*. _ will place you to the beginning of the line, *"a* will tell **yank** to use register `a`, and *y$* will copy the text until the end of the line.

That's it. Paste the register with *"ap*, and place it back when you've finished editing using *_"ay$*. Post this sentence online out of context to scare people from ever buying this book.

> **Tip**
>
> As with many Vim commands, you shouldn't try to remember the exact letters but focus on what the command does. The *_"ay$* command, for instance, goes to the beginning of the line and yanks the rest of the line into register `a`. That's much easier to remember than *_"ay$*.

Recursive macros

Earlier we ran macros multiple times by prefixing @ with a number. That's not very computer science-like, and we can do better.

Vim supports recursive macros, but there are a few quirks to be aware of.

First, you'll need to make sure the register you will be recording to is empty. You can do this by entering macro recording mode, and immediately exiting it. For instance, if you wanted to empty register `b`, you'd run *qbq* to clear it.

Then, record your macro as usual, and insert that same register into itself (for example, by using *@b*).

Let's say we wanted to swap keys with values in a Python dictionary. We could record a macro as follows, starting with the cursor at the beginning of the line `'egg': 'spam omelet'`:

```
dish_names = [
    'egg': 'spam omelet',
    'sausage': 'spam sausage',
    'bacon': 'bacon and spam'
]
```

Figure 6.17 – A dictionary where we'd want to swap keys and values.

Let's record a macro into the register `b`. First, we'll need to flush it and enter macro recording mode: *qbqqb* (*qbq* empties register `b`, and *qb* enters macro recording mode).

Now, as we want to swap `egg` and `spam omelet`, we could yank one of these words into some temporary register (say `c`), and then move `egg` over using the default register.

Let's go ahead: *"cdi'* (delete inside single quotes into register `c`):

```
dish_names = [
    '': 'spam omelet',
    'sausage': 'spam sausage',
    'bacon': 'bacon and spam'
]
recording @b
```

Move one **WORD** over (*W*) and run *di'* to yank `spam omelet` into the default register:

```
dish_names = [
    '': '',
    'sausage': 'spam sausage',
    'bacon': 'bacon and spam'
]
recording @b
```

Move one character to the left (either with *h* or *b*) and insert `egg` from register `c` (*"cp*):

```
dish_names = [
    '': 'egg',
    'sausage': 'spam sausage',
    'bacon': 'bacon and spam'
]
recording @b
```

Now, move back to the beginning of the line (_) and paste `spam omelet` from the default register (*p*):

```
dish_names = [
    'spam omelet': 'egg',
    'sausage': 'spam sausage',
    'bacon': 'bacon and spam'
]
recording @b
```

Almost there! Go down one line (*j*), and move your cursor to the beginning of the line (_):

```
dish_names = [
    'spam omelet': 'egg',
    'sausage': 'spam sausage',
    'bacon': 'bacon and spam'
]
recording @b
```

Now, replay macro `b`: *@b*. Nothing will happen, since register `b` is still empty. Finish recording the macro (`q`).

Now replay the macro using *@b* once, and it will iterate through every line in your file:

```
dish_names = [
    'spam omelet': 'egg',
    'spam sausage': 'sausage',
    'bacon and spam': 'bacon'
]
```

That's it! You can make any macro recursive by appending it to the register. To append to a register, you use the uppercase version of the register identifier. For example, if you wanted to make a macro in a register `b` recursive, you'd run *qB@bq* to append `@b` to the end of the macro.

Running macros across multiple files

If you wanted to replay a macro across multiple files, you could use (you guessed it) arglist. Arglist allows you to execute normal mode commands with the `:normal` command. For instance, you could run a macro from register `a`, as follows:

```
:arg **/* .py
:argdo execute ":normal @a" | update
```

Here, `:normal @a` will execute macro `a` in normal mode, and `update` will save the buffer contents. You'll probably want to use arglist with recursive macros.

Using plugins to do the job

"Wait," you exclaim. "There were plugins to do this all along?" Indeed, there are plugins that support refactoring operations—be it modifying parameters, renaming, or method extraction.

However, when working with existing refactoring solutions, I always find they do almost, but not quite, what I need. That's why I continue to write fancy substitute commands for refactoring. I find the cost of incorporating a refactoring plugin into my workflow, only to switch to `:substitute` commands for some of my refactoring needs, to be too high.

At the time of writing this book, there's no go-to refactoring plugin. Some are language-specific, and some focus on only certain aspects of refactoring. For instance, plugins like **YouCompleteMe** provide semantically aware renaming commands (such as `:YcmComplete RefactorRename`).

Your best bet is to figure out for yourself which operations you want to perform and try out a few plugins based on that. A web search along the lines of "`Vim refactoring plugins`" should do the trick.

Summary

In this chapter, we covered the `:substitute` command and macros—two powerful tools we can use for refactoring.

We covered the `:substitute` command and its flags. We looked into arglist, which is a way to execute a command across multiple files.

The `:substitute` command also supports regular expressions, which make your life a lot easier by allowing you to go beyond literal matches. We covered the basics of regular expressions and Vim magic modes (which are ways of interpreting special characters when parsing regex-enabled strings).

Finally, we looked at macros: a feature that lets you record and later replay a set of keystrokes. Macros can be edited the same way that registers can, and can also be made recursive to play as many times as needed.

In the next chapter, we'll cover customizing Vim for a personalized editing experience.

7
Making Vim Your Own

This chapter will cover Vim customization, and how to make Vim work for you. Everyone's needs are different, and this chapter tries to help you develop your own style.

This chapter will cover the following topics:

- Color schemes and making your Vim look pretty
- Enhancing the status line with additional information
- GUI configuration specific to gVim
- Healthy habits when customizing your workflow
- Methodologies for organizing your `.vimrc`

Technical requirements

In this chapter, we will be covering ways to keep your `.vimrc` file organized. There's no supporting code – you're welcome to bring along your `.vimrc` file and try out the techniques suggested in this chapter.

In addition, we'll be installing some packages with `pip`, so you may want to make sure you have pip installed. You can install pip by running:

```
$ python3 -m ensurepip --upgrade
```

Or alternatively, if the above doesn't work for some reason:

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py && \
  python3 get-pip.py
```

Playing with the Vim UI

Vim has an extensible UI, and it doesn't always have to look like it's stuck in the 90s. You can change its themes, tweak the way certain UI elements are displayed, and enhance the information displayed in a status line. If you're a gVim user, there are even more customization options available to you!

Color schemes

Vim has a plethora of beautiful color schemes available, both packaged with Vim and made by community members.

You can change the color scheme by changing the `colorscheme` setting in your `.vimrc`, as follows:

```
:colorscheme elflord
```

To get a list of currently installed color schemes, execute `:colorscheme` followed by *Ctrl* + *d*. This will list every installed color scheme:

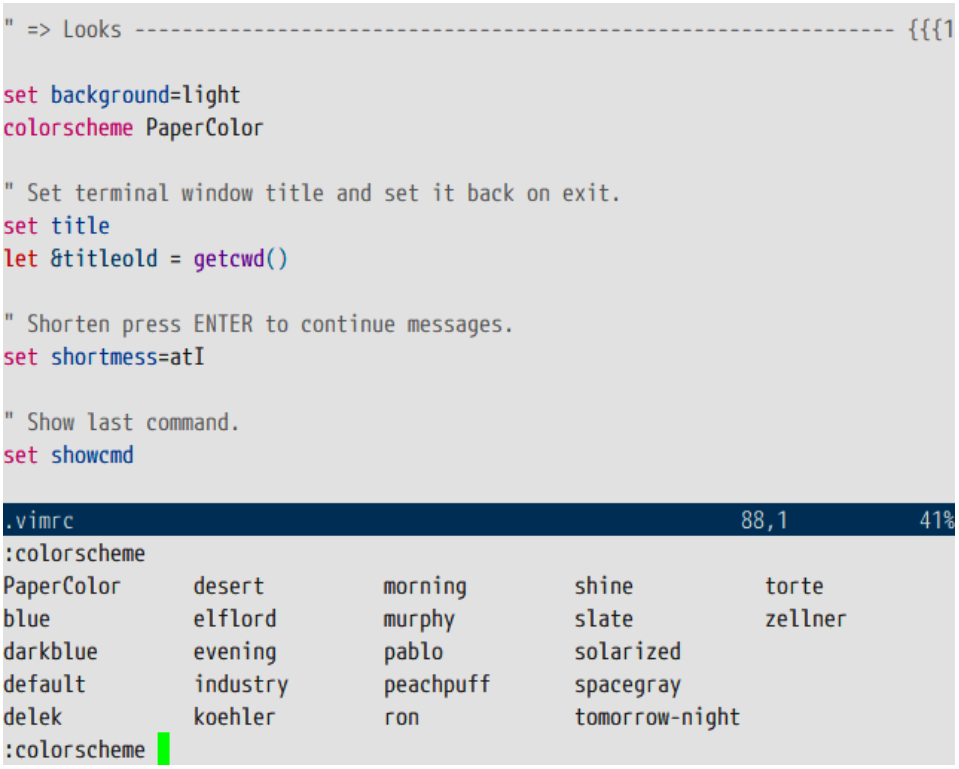

Figure 7.1 – Output of :colorscheme followed by *Ctrl* + *d*. Current color scheme is **PaperColor**.

In the preceding example, I'm using `:colorscheme PaperColor` from `https://github.com/NLKNguyen/papercolor-theme`.

You can further customize the color scheme by setting the `background` option to either light or dark (the option must precede the `colorscheme` call).

For example, this is how the same color scheme as in the preceding screenshot (`PaperColor`) looks with `set background=dark`:

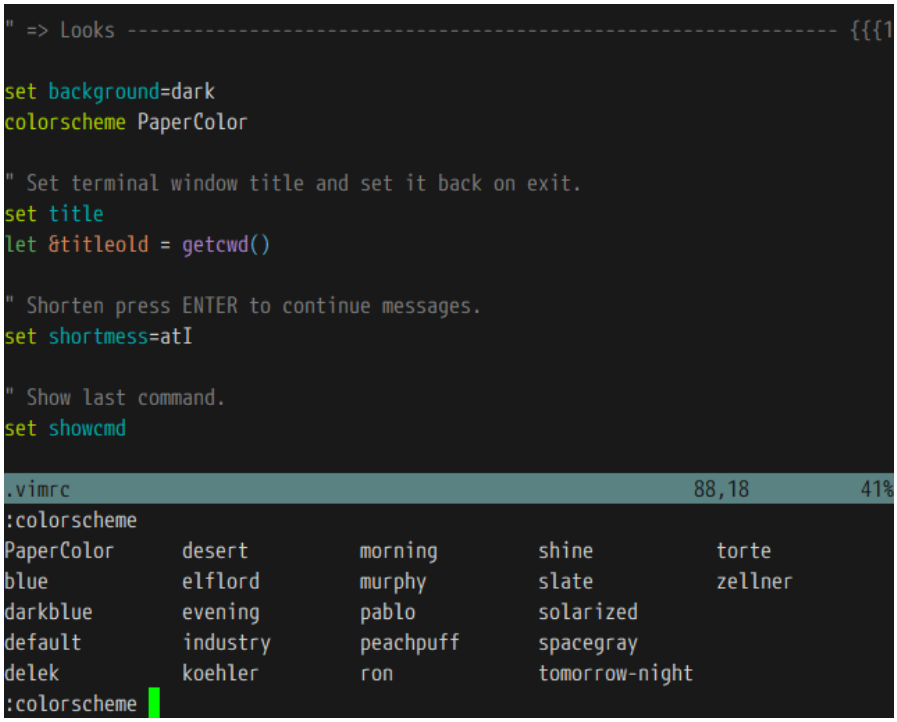

Figure 7.2 – **PaperColor** color scheme with `:set background=dark` option.

Browsing the color schemes

There are many color schemes available online, since tastes differ so significantly. There's no single indisputably authoritative resource for color schemes. Your best bet is to look around and try to find what catches your eye.

If you manage to get your hands on lots of color schemes while you're trying to find the one you like, you can use a helpful plugin called `ScrollColors`. Now, `ScrollColors` adds a `:SCROLL` command that lets you interactively cycle through your color schemes.

Install ScrollColors with vim-plug

If you're using vim-plug, you can install `ScrollColors` by adding `Plug 'vim-scripts/ScrollColors'` to your `.vimrc` and running `:w | source $MYVIMRC | PlugInstall`.

There's also a collection of color schemes available at `https://github.com/flazz/vim-colorschemes`, with what looks like a few hundred of the most popular color schemes. I found all of my personal favorites in that list, so it might be a good resource for someone who's trying to decide on a few favorite color schemes.

This plugin and `ScrollColors` can be used together to browse through a gallery of the most popular color schemes.

Install vim-colorschemes with vim-plug

`vim-colorschemes` can be installed with vim-plug by adding `Plug 'flazz/vim-colorschemes'` to your `.vimrc` and running `:w | source $MYVIMRC | PlugInstall`.

Common issues

Sometimes you'll find that the color schemes you're trying out don't look as nice or don't display as many colors as you see in screenshots online.

This is most likely because your Terminal emulator mistakenly tells Vim it only supports 8 colors, and not the 256 available in modern Terminal emulators. For that, you'll have to properly set the `$TERM` environment variable.

This is most common when using **tmux** and **GNU Screen**, as they might wrongly report the number of colors available.

Even more colors!

If you find 256 colors to not be enough, certain Terminals support 24-bit colors, often referred to as "truecolor." If your Terminal supports 24-bit truecolor (a quick web search would help), add `set termguicolors` to your `~/.vimrc`.

To view the current content of the `$TERM` environment variable, run the following in your shell:

```
$ echo $TERM
```

If you're using **tmux**, add the following to your `.tmux.conf`:

```
set -g default-terminal "xterm-256color"
```

If you're a **GNU Screen** user, add the following to your `.screenrc`:

```
term "xterm-256color"
```

If the preceding didn't apply to you, add the following to your `.bashrc`:

```
TERM=xterm-256color
```

However, overriding `$TERM` in your `.bashrc` is rarely a good idea, and you may want to do some deeper research into what sets your `$TERM` environment variable to the wrong format.

The status line

The status line is that lovely bar at the bottom of the screen that is used to display information. You can make it even more useful with some minor configuration tweaks:

```
" Always display a status line (it gets hidden sometimes otherwise).
set laststatus=2

" Show last command in the status line.
set showcmd
```

If you want to go even further, there are plugins to enhance your status line. Powerline is an everything-in-one powerhouse, while Airline is its lighter alternative.

Powerline

Powerline provides an enhanced status line for Vim, as well as providing other functions, such as extending your shell prompt or tmux status line. It is available (along with detailed installation instructions) from `https://github.com/powerline/powerline`. When enabled in Vim, it looks something like this:

Figure 7.3 – The **Powerline** status bar.

As you can see, it displays a plethora of information, including current mode, Git branch, filename, status of a current file, file type, encoding, and how far along you are in a current file. It's fully customizable, and lets you display as much or as little information as you want.

It's a bit of trouble to install, since it's not just a Vim plugin. First, you'll need to install the `powerline-status` package through pip:

```
$ python3 -m pip install powerline-status
```

If you don't have pip installed, see the technical requirements at the beginning of this chapter for setup instructions.

> **Note**
>
> To use Powerline, you need a Vim build that includes a python3 interpreter. That is not always the case and may require you to install (or even self-compile) a different Vim flavor. Check the output of `:version` for `+python3`.

You'll also need to make sure `$HOME/.local/bin` (the default scripts location for pip) is on your path by adding the following to your `.bashrc`:

```
PATH=$HOME/.local/bin:$PATH
```

Finally, set `laststatus` to `2` (to make sure the status line is always displayed), and load Powerline in your `.vimrc`:

```
" Always display status line (or what's the purpose of having
powerline?)
set laststatus=2

" Load powerline.
python3 from powerline.vim import setup as powerline_setup
python3 powerline_setup()
python3 del powerline_setup
```

Now, reload your Vim configuration (`:w | source $MYVIMRC`), and you'll see the fancy new status line at the bottom of your screen, displaying current mode (`NORMAL`), open file (`~/.vimrc`), file format (`unix`), encoding (`utf-8`), file type (`vim`), how far down your screen is scrolled (`2%`), and selected line range (`1:1`):

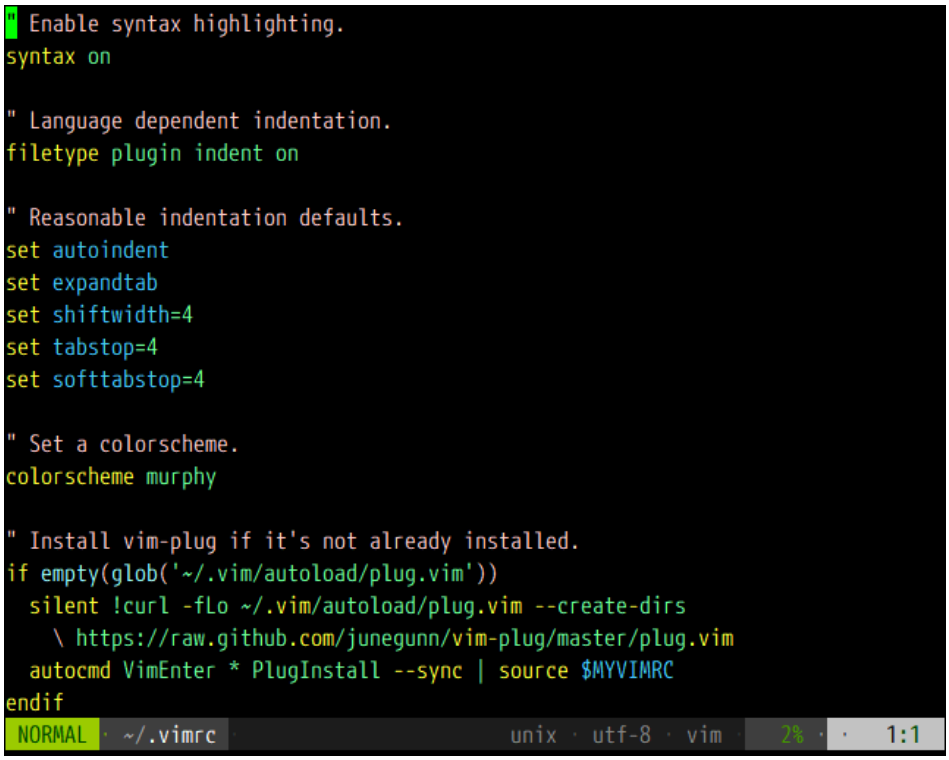

Figure 7.4 – The **Powerline** plugin.

Airline

Airline is a great alternative if you don't want anything extra, and don't like the idea of a Python daemon continuously running in the background. It provides an informative, nice-looking prompt, as in the following screenshot:

Figure 7.5 – The **Airline** plugin.

Airline is available from `https://github.com/vim-airline/vim-airline`, and has no additional dependencies.

> **Install vim-airline with vim-plug**
>
> You can install Airline with vim-plug by adding `Plug 'vim-airline/vim-airline'` to your `.vimrc` and running `:w | source $MYVIMRC | PlugInstall`.

gVim-specific configuration

gVim is a standalone application, and lets you configure more than out-of-the-box Vim does. In fact, gVim supports having its own configuration file (in addition to `.vimrc`): `.gvimrc`.

The primary option for managing how the GUI looks is `guioptions`. This configuration string takes a set of letters, which enable options. Some relevant settings might include the following:

- `a` and `P`—automatically yank the visual selection into the system clipboard (for `*` and `+` registers respectively, see *Registers* in *Chapter 2*, *Advanced Editing and Navigation*)
- `c`—use console dialogs instead of popups
- `e`—display tabs using GUI components

- `m`—display a menu bar
- `T`—include a toolbar
- `r`, `l`, and `b`—make right, left, and bottom scrollbars always visible

> **Your favorite font**
>
> gVim also lets you set your preferred font by setting the `guifont` option. Read up on it by running `:help 'guifont'`.

For example, if you wanted to display a menu bar and a toolbar and always display a bottom scroll bar, you could do so by adding the following to your `.vimrc`:

```
" GUI: Enable menu bar, toolbar, always display bottom scrollbar.
set guioptions=mTb
```

The changes will look like this (this screenshot depicts gVim in Windows):

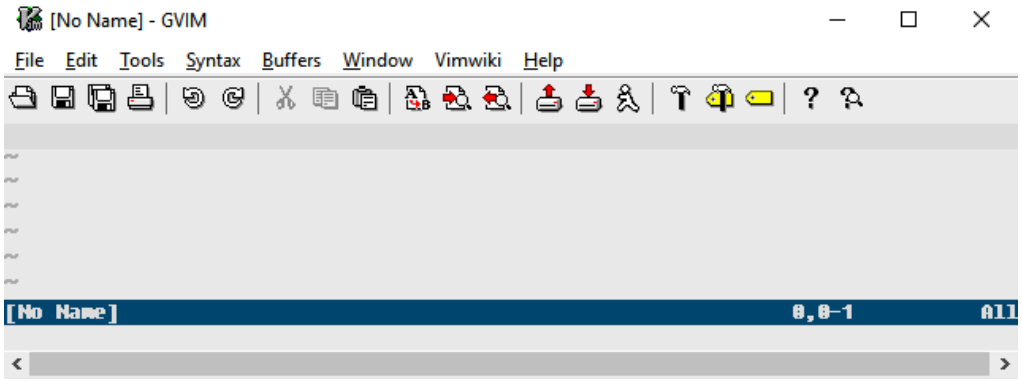

Figure 7.6 – GVim with m (menu bar), T (toolbar), and b (bottom scrollbar) options.

You can learn more about gVim-specific options by reading `:help gui`.

Keeping track of configuration files

Chances are, you won't spend the next ten years using the same computer. It's also possible you have multiple machines you work across—so you should probably find a way to synchronize your configuration files across multiple environments.

As usual, there's no single right way to do this, but a common practice is to store files in a Git repository (often called `dotfiles`, since configuration files in Linux tend to start with a dot), and pointing symbolic links (symlinks) from the files in the home directory to the files in the `dotfiles` directory.

All you'll have to do is commit, push, and pull the configuration with Git on each machine to stay up to date.

> **Symlinks**
>
> If you're not familiar with symlinks, they're essentially a pointer to a file. You can create a symlink using the `ln` command in your Terminal, e.g: `ln -s <original file> <link to file>`.

The easiest way would probably be to create a repository using a service such as GitHub, and utilize it to synchronize your configuration. Just don't store any sensitive information like passwords in version control!

Most frequently, the process of changing my configuration files is as follows (I store mine in `$HOME/.dotfiles` on Linux and Mac, and `%USERPROFILE%\_dotfiles` on Windows):

```
$ cd ~/.dotfiles
$ git pull --rebase
# Make the desired changes, like editing .vimrc
$ git commit -am "Updated something important"
$ git push
```

> **Git!**
>
> The `.dotfiles` should be a Git repository; see *Quick and dirty* in *Chapter 5*, *Build, Test, and Execute* to learn how to create one and if you'd like a refresher on Git.

For example, you might have a repository in `~/dotfiles`, which contains `.vimrc` and `.gvimrc` files, as well as a `.vim` directory. You could create links manually (`ln -s ~/dotfiles/.vimrc .vimrc`), or by writing a small Python script like this:

```
import os

dotfiles_dir = os.path.join(os.environ['HOME'], 'dotfiles')
for filename in os.listdir(dotfiles_dir):
    os.symlink(
        os.path.join(dotfiles_dir, filename),
        os.path.join(os.environ['HOME'], filename))
```

You can get infinitely creative with solutions to this problem. Here are just a few examples of where you can take this:

- Make the preceding Python script work cross-platform (for instance, the `.vim` directory becomes `vimfiles` in Windows)

- Periodically synchronize the Git repository using a cron job
- Use some form of file sync instead of Git (trading informative commit messages for near-instant updates across machines)

Healthy Vim customization habits

As you continue to work with Vim, you will find yourself making a lot of configuration changes. It's important to take time to go back, reflect, and make sure your `.vimrc` doesn't become a pile of unneeded aliases, functions, and plugins.

Once in a while, take the time to go back into your `.vimrc` and clean up unnecessary functions and plugins, or remove key bindings you don't use anymore. If you don't know what something does, you're probably better off removing it, since you won't get much use out of configurations you don't understand.

It's also helpful to take some time to read about the options you have set and plugins you have installed with the built-in `:help` command—you never know what useful feature you'll discover!

Optimizing your workflow

Everyone's workflow is unique, and no two people use Vim the same way. It's useful to find ways to complement your style by enhancing and optimizing the way you do things in Vim.

Do you find yourself using a particular command a lot? Create a custom key binding!

For example, I use the **CtrlP** plugin quite a lot (both for navigating the file tree and the buffer list), and I have the following custom mappings:

```
nnoremap <leader>p :CtrlP <cr>
nnoremap <leader>t :CtrlPTag <cr>
```

I also often find myself running the `:Ack` command (provided by the ack-vim plugin) on a word under cursor, so I have the following in my `.vimrc`:

```
nnoremap <leader>a :Ack! <c-r><c-w><cr>
```

`<c-r>` followed by `<c-w>` inserts the word under cursor into the command line.

Can we use `:grep` for a similar purpose? Not a problem:

```
nnoremap <leader>g :grep <c-r><c-w> */**<cr>
```

Accidentally find yourself hitting `;` instead of `:` to enter command-line mode? (I do all the time!) Remap it:

```
nnoremap ; :
vnoremap ; :
```

Whenever you catch yourself doing something a lot, take a moment to add a relevant key binding to make your life easier.

Keeping .vimrc organized

If you use and customize Vim a lot, your `.vimrc` file will tend to grow rather quickly, and it's important to make it easy to navigate. If you ever take a break from working with your `.vimrc` and come back later, you'll thank yourself.

Comments are crucial for making sure you remember what's going on. If you take just one thing from this chapter, make sure it is comments. Just like when working with code, comments save you from wasting time trying to understand what's going on.

I try to make a point of documenting every configuration bit with a corresponding comment:

```
" Show last command in the status line.
set showcmd

" Highlight cursor line.
set cursorline

" Ruler (line, column and % at the right bottom).
set ruler

" Display line numbers if terminal is wide enough.
if &co > 80
  set number
endif

" Soft word wrap.
set linebreak

" Prettier display of long lines of text.
set display+=lastline

" Always show statusline.
set laststatus=2
```

:help with options

You can always view a help page for any option through `:help '<option>'`, e.g., `:help 'laststatus'`.

Some people might prefer to place the comments on the same line as the configuration bits:

```
set showcmd " Show last command in the status line.

set cursorline " Highlight cursor line.

set ruler " Ruler (line, column and % at the right bottom).

if &co > 80 " Display line numbers if terminal is wide enough.
  set number
endif

set linebreak " Soft word wrap.

set display+=lastline " Prettier display of long lines of text.

set laststatus=2 " Always show statusline.
```

For plugins in particular, I find it extremely useful to add a quick comment explaining what each one of them does. This makes it easy to revise the list of plugins once I don't need certain ones anymore:

```
Plug 'EinfachToll/DidYouMean' " filename suggestions
Plug 'easymotion/vim-easymotion' " better move commands
Plug 'NLKNguyen/papercolor-theme' " colorscheme
Plug 'ajh17/Spacegray.vim' " colorscheme
Plug 'altercation/vim-colors-solarized' " colorscheme
Plug 'christoomey/vim-tmux-navigator' " better tmux integration
Plug 'ervandew/supertab' " more powerful Tab
Plug 'junegunn/goyo.vim' " distraction-free writing
Plug 'ctrlpvim/ctrlp.vim' " Ctrl+p to fuzzy search
Plug 'mileszs/ack.vim' " ack integration
Plug 'scrooloose/nerdtree' " prettier netrw output
Plug 'squarefrog/tomorrow-night.vim' " colorscheme
Plug 'tomtom/tcomment_vim' " commenting helpers
Plug 'tpope/vim-abolish' " change case on the fly
Plug 'tpope/vim-repeat' " repeat everything
Plug 'tpope/vim-surround' " superchange surround commands
Plug 'tpope/vim-unimpaired' " pairs of helpful shortcuts
Plug 'tpope/vim-vinegar' " - to open netrw
Plug 'vim-scripts/Gundo' " visualize the undo tree
Plug 'vim-scripts/vimwiki' " personal wiki
```

There are many ways to make configuration easier to navigate. My organizational method of choice is marker folds. I break down my configuration into categories, such as *looks*, *editing*, or *movement and search*. Then I use manual fold markers (`{{{1`) to indicate folds.

I also tend to add some ASCII art in the form of arrows `=>` and lines `---` to make each section look more like a header:

```
...

" => Editing ---------------------------------------------- {{{1
syntax on

...

" => Looks ------------------------------------------------ {{{1

set background=light
colorscheme PaperColor

...
```

This way, if I want an overview of my `.vimrc` file, I can close all folds using *zM*, and I'll get a helpful view:

```
" URL: https://github.com/ruslanosipov/dotfiles
" Author: Ruslan Osipov
" Description: Personal .vimrc file
"
" All the plugins are managed via vim-plug, run :PlugInstall to install all
" the plugins from Github, :PlugUpdate to update. Leader key is the spacebar.
"
" What function keys do (also see: Custom commands, Leader shortcuts):
"   F5: toggle Gundo window.

+-- 16 lines: => Pre-load ---------------------------------------------------
+-- 27 lines: => vim-plug plugins ---------------------------------------------
+-- 41 lines: => Editing ------------------------------------------------------
+-- 35 lines: => Looks ---------------------------------------------------------
+--  5 lines: => Custom commands ---------------------------------------------
+-- 11 lines: => Leader shortcuts ---------------------------------------------
+-- 16 lines: => Movement and search ------------------------------------------
+-- 20 lines: => Filetype-specific --------------------------------------------
+-- 14 lines: => Misc ---------------------------------------------------------
+-- 21 lines: => Fixes and hacks ----------------------------------------------
+-- 17 lines: => Plugins configuration ----------------------------------------
~
.vimrc                                                1,1           All
".vimrc" 233L, 6275C [w]
```

Figure 7.7 – My personal `.vimrc` file using manual folds.

Other ways to organize your configuration

How you organize your configuration is completely subjective. For example, you could group similar customizations into their own files and source them individually: `mappings.vim` for mappings, `functions.vim` for custom functions, `plugins.vim` for all the plugin settings, etc.

Summary

In this chapter, we've covered ways to enhance Vim's user interface and personalize Vim.

We've looked at color schemes, ways to configure them, finding them, and browsing them. We've also looked at ways to enhance Vim's status line with the heavyweight **Powerline** or lightweight **Airline** plugin.

We've looked at GUI configuration specific to gVim, and how to customize the way gVim looks.

Finally, as you use Vim more, you'll develop your own style and personal workflow. This workflow is best enhanced by bindings and shortcuts. As your `.vimrc` grows, there are several ways to get it organized, well documented, and easy to navigate.

In the next chapter, we'll learn about Vimscript, an extensive scripting language that comes packaged with Vim.

8

Transcending the Mundane with Vimscript

This chapter will cover Vimscript in all its glory. We will go into quite a bit of detail, but since we only have so many pages, the coverage will be somewhat spotty. Hopefully, this chapter will get you interested in Vimscript enough to start your own research, and maybe you can use it as a reference as you build your early plugins and scripts. In this chapter, we will look at the following:

- The basic syntax, from declaring variables to using lambda expressions
- Style guides, and how to keep sane when developing in Vimscript
- A sample plugin from start to finish—from the first line to publishing it online

Technical requirements

This chapter walks you through learning Vimscript by using numerous examples. All the examples are available on GitHub: `https://github.com/PacktPublishing/Mastering-Vim-Second-Edition/tree/main/Chapter08`. You can create the scripts on your own as we're working through this chapter, or download the files from the repository to play around with.

Why Vimscript?

You've already encountered Vimscript when you worked on your `.vimrc` file. What you may not have known is that Vimscript is actually a Turing-complete scripting language—there's no limit to what you can do. So far, you've used it to set variables and perform a few comparison operations, but it can do so much more!

You will learn how Vimscript not only helps you understand Vim configuration better but also lets you solve text editing problems you encounter by writing functions and plugins.

It's pretty awesome.

How to execute Vimscript

Vimscript is made up of commands you run in command-line mode and really is just a sequence of Vim commands in a file. You can always execute Vimscript by running each command in command mode (the one you prefix with `:`), or by executing the file with commands using a `:source` command. Historically, Vim scripts have a `.vim` extension.

As you're following along with this section, you may want to create `*.vim` files to experiment in. You can execute the files by running this:

```
:source <filename>
```

A much shorter version of that is this:

```
:so %
```

Here, `:so` is a short version of `:source`, and `%` refers to the currently open file.

For example, I just created a `variables.vim` file to play around with Vim's variables. I could execute its contents with `:so %`:

Figure 8.1 – Output of the :so % command

Alternatively, I could run each command in command mode. For example, if I wanted to print the contents of a variable, `g:ingredient`, I would run the following:

```
:echo g:ingredient
```

I will do just that, as in, print `egg` into our status line:

```
let g:ingredient = 'egg'

echo 'Scene: A cafe. A man and his wife enter.'
echo 'Man: Well, what''ve  you got?'
echo g:ingredient
echo '- said the waitress'
~
~
~
~
tutorial/01_variables.vim                                   1,1           All
egg
```

Figure 8.2 – Output of `:echo g:ingredient`

Normally, I run longer scripts with `:so %`, and perform debugging operations through command-line mode (`:`).

Additionally, if you're entering commands in command-line mode, you'll stay in command-line mode if you enter a function or a flow control operator (such as `if`, `while`, or `for`):

```
tutorial/01_variables.vim                                   1,1           All
:if has('win32')
:  echo 'this is windows'
:  else
:  echo 'this is probably linux'
this is probably linux
:  endif
```

Figure 8.3 – Flow control operators in command-line mode

In this example, I did not have to type `:` on every line. Additionally, each line gets executed as you hit *Enter* (as you can see by `this is probably linux` being printed on the screen).

> **Tip**
>
> You can also use the `|` (pipe) separator to make the statement in-line: `if has('win32')|echo 'this is windows' | else | echo 'this is probably linux' | endif`.

Major changes in Vimscript 9

Vim 9 introduced **Vimscript 9** (or **Vim9script**), which significantly improves performance and gives users access to programming constructs available in common programming languages. Vimscript 9 is best treated as an addition to Vimscript, to be used alongside prior versions of Vimscript.

Outside of improved performance, differences come down to syntax – for example, signifying comments with # instead of `"`, declaring variables with `var`, explicit boolean support, required use of whitespace for readability, and so on. I'll call out the differences between Vim9script and previous versions of Vimscript throughout this chapter.

Vimscript 9 use is opt-in and can be invoked by starting your script with `vim9script`, defining functions through the `def` keyword (rather than `function`), or prefixing a command with `vim9cmd`.

I recommend using Vimscript 9 if you intend to get more serious about writing custom plugins or start fiddling with performance-sensitive functions. You could even convert your `.vimrc` file to Vim9script, but you'd lose backward compatibility (which might or might not be a deal-breaker for you, depending on how you use Vim).

Throughout this chapter, I will default to legacy Vimscript and will call out differences in Vim9script.

Learning the syntax

Let's take a lightning-fast deep dive into Vimscript's syntax.

> **Prior knowledge required!**
>
> This section assumes you're comfortable with at least one programming language, conditional statements, and loops in particular. If that's not the case, you will most certainly be better off finding a more extensive tutorial. Vimscript deserves its own book, and Steve Losh wrote just that: *Learn Vimscript the Hard Way* is easily the best Vimscript tutorial available (and it's free on the web!).

Setting variables

You've already discovered some basics of Vimscript syntax. To set internal Vim options, you use the `set` keyword:

```
set background=dark
```

> **Reminder**
>
> Don't forget, you can run `:source %` to execute the current file.

To define a variable, you can use `let` (for variables):

```
let dish = 'spam omelet'
```

To assign a value to a non-internal variable, you'll again use the `let` keyword:

```
let dish = 'bacon and spam'
```

> **Assigning variable values in Vim9script**
> In Vim9script, you use `var`, `const`, and `final` to declare a variable. Omit `let` when changing variable value (e.g., `dish = 'bacon and spam'`).

Legacy Vimscript doesn't have explicit booleans, so `1` is treated as true and `0` as false:

```
let has_spam = 1
```

> **Vim9script supports explicit booleans**
> Vim9script supports explicit `true` and `false` values, allowing you to use a construct such as `var has_spam = true`.

Since we're assigning variables, let's talk about scopes. Vim handles variable and function scopes with prefixes, like so:

```
let g:dish = 'spam omelet'
let w:has_spam = 1
```

Each letter has a unique meaning, in particular the following:

- `g`: global scope (default if scope is not specified, unless in a function)
- `v`: global defined by Vim
- `l`: local scope (default within a function if scope is not specified)
- `b`: current buffer
- `w`: current window
- `t`: current tab
- `s`: local to a `:source'd` Vim script
- `a`: function argument

In this example, `g:dish` is a global variable. By default, in legacy Vimscript, the variables are in global scope. It could also be written as `let dish = 'spam omelet'`), and `w:has_spam` is a window scope variable.

Scopes in Vim9script

In Vim9script, the default variable scope is the current script (`s:`), which certainly makes more sense. Vim9script also omits the `a:` declaration from function arguments, instead making arguments part of the local (`l:`) scope.

As you might remember, you can also set registers with `let`. For example, if you wanted to set register `a` to hold `spam spam spam`, you could do this:

```
let @a = 'spam spam spam'
```

You can also access Vim options (the ones you can change with `set`) by prefixing the variable with `&`, as in this example:

```
let &ignorecase = 0
```

You can use the usual mathematical operations on integers (`+`, `-`, `*`, and `/`). String concatenation is performed using the `.` operator:

```
const g:statement = 'Well, we''ve got ' . g:dish
```

String concatenation in Vim9script

In Vim9script, concatenation is instead performed with the `..` operator: for example, `const g:statement = 'Well, we''ve got ' .. g:dish`.

If you want to use a single quote within a single-quoted string, you can do so by typing it twice (`''`).

Oh, and like in many languages, single quotes identify literal strings, while double quotes identify non-literal strings. This becomes slightly confusing because comments also start with a double quote. Due to this behavior, certain commands in Vimscript can't be followed by a comment.

Speaking of comments in Vim9script

In Vim9script, comments are signified with #, just like in Python.

Surfacing output

You can print the content of a variable (or the results of any operation) into a status line using `echo`:

```
echo dish
```

One thing about echo, though, is that the output does not get logged anywhere and there's no way to view the message once it's dismissed.

For that, there's `:echomsg` (or `:echom` for short):

```
echom 'Well, we''ve got ' . dish
echom 'here is another message'
```

To see the log of messages from the current session, execute the following command:

```
:messages
```

Now, you can see every message we printed:

```
echo dish

echom 'Well, we''ve got ' . dish
echom 'here is an another message'
~
02_output.vim                                          1,0-1         All
Messages maintainer: The Vim Project
"02_output.vim" 9L, 107B
Well, we've got egg omelet
here is an another message
Press ENTER or type command to continue
```

Figure 8.4 – Outputting messages with :messages

In fact, many operations log messages via `echom`. For instance, a file write using `:w` does so:

```
echom 'Well, we''ve got ' . dish
echom 'here is an another message'
~
02_output.vim                                          2,0-1         All
Messages maintainer: The Vim Project
"02_output.vim" 9L, 107B
Well, we've got egg omelet
here is an another message
"02_output.vim" 9L, 107B [w]

Press ENTER or type command to continue
```

Figure 8.5 – Writes also show up in :messages

Messages can be a powerful tool for debugging and trying to figure out what went wrong with your script. Learn more about messages via `:help message-history`.

Conditional statements

Conditional statements are performed using `if` statements:

```
if ingredient == 'egg'
  echo 'spam omelet'
elseif ingredient == 'lobster'
  echo 'spam lobster thermidor'
else
  echo dish . ' and spam'
endif
```

The `if` statement is also available inline by using `?` and `:` operators:

```
echo 'spam ' . (ingredient == 'egg' ? 'omelet' : dish)
```

Vim supports all the logical operators you're used to from other languages:

- `&&` - and
- `||` - or
- `!` - not

For example, you can do this:

```
if !(is_egg || is_lobster)
  echo ingredient . ' and spam'
endif
```

In the previous example, you'll get to `echo ingredient . ' and spam'` only if `ingredient` isn't an egg or a lobster.

This can also be written with the `&&` operator:

```
if !is_egg && !is_lobster
  echo ingredient . ' and spam'
endif
```

Since text editing implies operating on strings, Vim has additional text-specific comparison operators:

- `==` compares two strings; case sensitivity depends on the user's settings (see later)
- `==?` explicitly case-insensitive comparison
- `==#` explicitly case-sensitive comparison

- `=~` checks a match against an expression on the right (`=~?` or `=~#` to make those explicitly case-insensitive or sensitive)
- `!~` checks that a string does not match an expression on the right (`!~?` or `!~#` to make those explicitly case-insensitive or sensitive)

The default behavior of `==`, as well as `=~` and `!~` (case-sensitive or case-insensitive) depends on the `ignorecase` setting. In fact, that is the reason (and, therefore, recommended) to always be explicit and use the `=~?` or `=~#` comparison operators to be independent of the users' setting.

Here are some examples:

```
'egg' ==? 'EGG'                  " true
'egg' ==# 'EGG'                  " false
set ignorecase | 'egg' == 'EGG' " true
'egg' =~ 'e.\+'                  " true
'egg' =~# 'E.\+'                 " false
'egg' !~ '.gg'                   " false
'egg' !~? 'E.\+'                 " false
```

Lists

Vim also supports more complex data structures, such as lists and dictionaries. Here's an example of a list:

```
let ingredients = ['egg', 'bacon', 'sausage']
```

Operations to modify lists are similar to the ones in Python. Let's take a look at some common operations.

You can get elements by index using the `[n]` syntax. Here are some examples:

```
let egg = ingredients[0]     " get first element
let bacon = ingredients[1]   " get second element
let sausage = ingredients[-1] " get last element
```

Slices work in a similar way to Python, as in this example:

```
let slice = ingredients[1:]
```

The value of `slice` would be `['bacon, 'sausage']`. The main difference from Python is that the end of the range is inclusive:

```
let slice = ingredients[0:1]
```

The value of `slice` would be `['egg', 'bacon']`.

To append to the list, use `add`:

```
call add(ingredients, 'lobster')
```

> **Function calls in Vimscript versus Vim9script**
> In legacy Vimscript, we explicitly call functions with `call` unless they're part of an expression. In Vim9script, the `call` keyword is omitted.

This will turn our list into `['egg', 'bacon', 'sausage', 'lobster']`. While it's an in-place operation, it also returns the modified list, so you can assign to it as well:

```
let ingredients = add(ingredients, 'lobster')
```

You can also prepend to the list using `insert`:

```
call insert(ingredients, 'tomato')
```

This will modify the list to be `['tomato', 'egg', 'bacon', 'sausage', 'lobster']`.

Lists in Vimscript are zero-based and you can also provide an optional index argument. For instance, if you wanted to insert `'ham'` at index `2` (where `'bacon'` currently is) in the previous list, you would do the following:

```
call insert(ingredients, 'ham', 2)
```

The list will become `['tomato', 'egg', 'ham', 'bacon', 'sausage', 'lobster']`.

There are a few ways to remove the elements. For example, you can use `unlet` to remove an element at index `2` (`'ham'`):

```
unlet ingredients[2]
```

The list will be back to `['tomato', 'egg', 'bacon', 'sausage', 'lobster']`.

You can also use `remove`:

```
call remove(ingredients, -1)
```

This will leave us with `['tomato', 'egg', 'bacon', 'sausage']`.

Additionally, `remove` also returns the item itself:

```
let tomato = remove(ingredients, 0)
```

You can also use ranges with both `unlet` and `remove`. Here's an example of deleting everything up to and including the second element:

```
unlet ingredients[:1]
```

If you were to do this with `remove`, you'd have to specify boundaries explicitly:

```
call remove(ingredients, 0, 1)
```

You can concatenate the lists using + or `extend`. Here is an example, given the `mammals` and `birds` lists:

```
let fresh = ['egg', 'lobster']
let preserved = ['bacon', 'sausage']
```

We could create a new list:

```
let ingredients = fresh + preserved
```

Here, `ingredients` will contain `['egg', 'lobster', 'bacon', 'sausage']`. We could also extend the existing list:

```
call extend(fresh, preserved)
```

This will extend `fresh` to contain `['egg', 'lobster', 'bacon', 'sausage']`.

You can sort the list in place using `sort`. If we were to use sort on a previous example, we would write the following:

```
call sort(ingredients)
```

The result would be `['bacon', 'egg', 'lobster', 'sausage']` (sorted alphabetically).

You can get an index of an element using `index`. For instance, if you wanted to get an index of `lobster` from the previous list, you would run this:

```
let i = index(ingredients, 'lobster')
```

In this case, `i` would be equal to 2.

You can check whether a list is empty using (aptly named) `empty`:

```
if empty(ingredients)
  echo 'There are no ingredients!'
endif
```

The length of a list is retrieved using `len`:

```
echo 'There are ' . len(ingredients) . ' ingredients.'
```

Finally, Vim lets you count the number of elements in a list:

```
echo 'There are ' . count(ingredients, 'egg') . ' eggs.'
```

> **:help**
> You can get a full list of operations by checking out the help page: `:help list`.

Dictionaries

Dictionaries are also supported in Vim:

```
let menu = {
  \ 'egg': 'spam omelet',
  \ 'bacon': 'bacon and spam',
  \ 'sausage': 'spam with sausage'
  \ }
```

As you may have noticed, you need to explicitly outline the line breaks with a backslash, \, if you're defining a dictionary on multiple lines.

> **Newlines in Vim9script**
> Unlike in legacy Vimscript, in Vim9script, line continuation does not always require a backslash.

Dictionary modification operations are similar to the ones familiar to you from Python. Elements can be accessed in two ways:

```
let egg_dish = menu['egg']  " get an element
let egg_dish = menu.egg     " another way to access an element
```

Accessing an element via `.` only works if the key contains numbers, letters, and underscores.

You can set or override a dictionary entry as follows:

```
let menu['lobster'] = 'lobster thermidor'
```

Entries are removed using `unlet` or `remove`:

```
unlet menu['lobster']
let lobster = remove(menu, 'lobster')
```

Dictionaries can be merged using `extend` (in place):

```
call extend(menu, {'lobster': 'lobster thermidor'})
```

This will make `menu` look as follows:

```
{
  \ 'egg': 'spam omelet',
  \ 'bacon': 'bacon and spam',
  \ 'sausage': 'spam with sausage',
  \ 'lobster': 'lobster thermidor'
  \ }
```

In case the second argument to `extend` contains duplicate keys, the original entries will be overwritten. This is configurable using the optional third parameter of `extend`.

Similarly to lists, you can measure dictionary length and check whether the dictionaries are empty:

```
if !empty(menu)
  echo 'There are ' . len(menu) . ' dishes in the menu.'
endif
```

Lastly, you can check whether a key is present in a dictionary using `has_key`:

```
if has_key(menu, 'egg')
  echo 'An egg dish is called ' . menu['egg']
endif
```

You can get a full list of operations by checking out the help page: `:help dict`.

Loops

You can loop through lists and dictionaries using the `for` keyword. For example, do this to go through a list:

```
for ingredient in ['egg', 'bacon', 'sausage']
  echo ingredient
endfor
```

And here's you iterating through a dictionary from a previous section:

```
for ingredient in keys(menu)
  echo 'A dish with ' . ingredient . ' is called ' . menu[ingredient]
endfor
```

You can also access both the key and the value of the dictionary simultaneously using `items`:

```
for [ingredient, dish] in items(menu)
  echo 'A dish with ' . ingredient . ' is called ' . dish
endfor
```

You can control the iteration flow with `continue` and `break`. Here's an example of using `break`:

```
let ingredients = ['egg', 'bacon', 'sausage']
for ingredient in ingredients
  if ingredient == 'bacon'
    echo 'Found bacon! Breaking!'
    break
  endif
  echo 'Looking at an ingredient ' . ingredient . ', no bacon yet.'
endfor
```

The output from this would be the following:

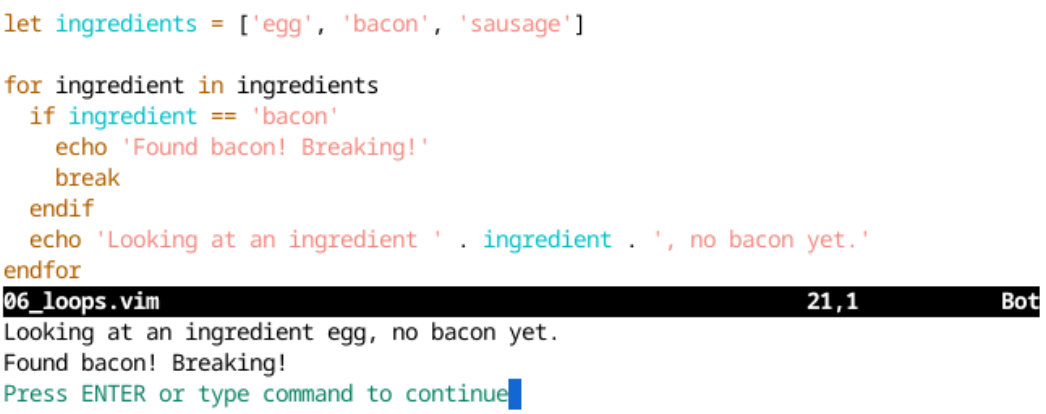

Figure 8.6 – How to use the break keyword in Vimscript for loops

And this is how you would use `continue`:

```
let ingredients = ['egg', 'bacon', 'sausage']
for ingredient in ingredients
  if ingredient == 'egg'
    echo 'Ignoring the egg...'
    continue
  endif
  echo 'Looking at an ingredient ' . ingredient
endfor
```

The output from this would be the following:

```
let ingredients = ['egg', 'bacon', 'sausage']

for ingredient in ingredients
  if ingredient == 'egg'
    echo 'Ignoring the egg...'
    continue
  endif
  echo 'Looking at an ingredient ' . ingredient
endfor
06_loops.vim                                                    31,1          Bot
Ignoring the egg...
Looking at an ingredient bacon
Looking at an ingredient sausage
Press ENTER or type command to continue
```

Figure 8.7 – How to use the continue keyword in Vimscript for loops

`while` loops are also supported:

```
let ingredients = ['egg', 'bacon', 'sausage']
while !empty(ingredients)
  echo remove(ingredients, 0)
endwhile
```

This will print the following:

```
let ingredients = ['egg', 'bacon', 'sausage']

while !empty(ingredients)
  echo remove(ingredients, 0)
endwhile
06_loops.vim                                                    37,1          Bot
egg
bacon
sausage
Press ENTER or type command to continue
```

Figure 8.8 – Example of a while loop

You can use `break` and `continue` the same way with `while` loops:

```
let ingredients = ['egg', 'bacon', 'sausage']
while len(ingredients) > 0
  let ingredient = remove(ingredients, 0)
```

```
  if ingredient == 'bacon'
    echo 'Found the bacon, breaking!'
    break
  endif
  echo 'Looking at an ingredient ' . ingredient
endwhile
```

This will output the following:

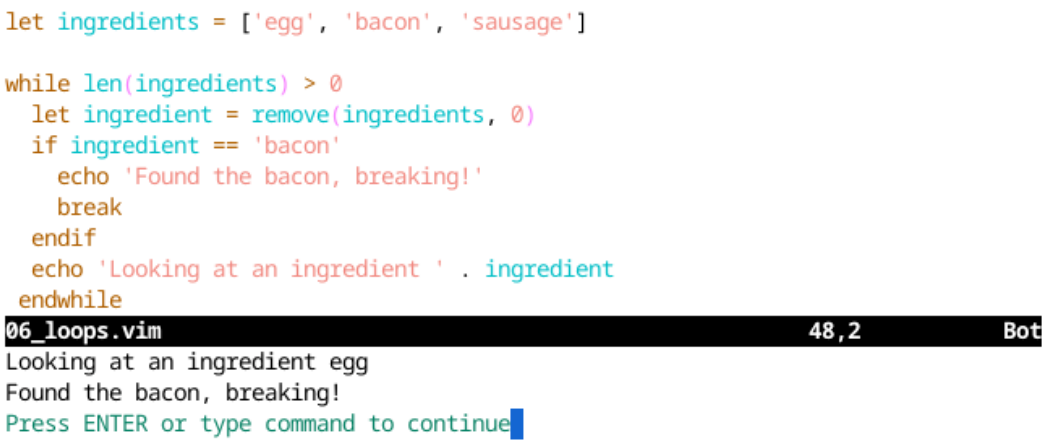

Figure 8.9 – Usage of a break keyword within a while loop

Functions

Just like most other programming languages, Vim supports functions:

```
function PrepareIngredient(ingredient)
  echo a:ingredient . ' and spam'
endfunction
```

Note that in legacy Vimscript, function arguments are part of the special `a:` scope.

> **Function names**
>
> In Vim, user-defined function names must start with a capital letter (unless they're within a script scope or behind a namespace). You'll get an error if you try to define a function starting with a lowercase letter.

You can try calling the function and you'll get the following output:

```
:call PrepareIngredient('sausage')
```

Figure 8.10 – Output of :call PrepareIngredient('sausage')

You can see that function arguments are accessed via the `a:` scope.

> **Functions in Vimscript 9**
>
> In Vim9script, functions are declared with the `def` keyword, and arguments are available in a local scope (`l:`) instead of an argument scope (`a:`). Further, Vimscript 9 requires you to specify the argument type, such as `PrepareIngredient(ingredient: string)`. If a function returns a value, you must specify a return type as well: `PrepareIngredient(ingredient: string): string`.
>
> Finally, you don't need to explicitly use the call keyword to call a function in Vimscript 9.

Functions, of course, can return values:

```
function PrepareIngredient2(ingredient)
  return a:ingredient . ' and spam'
endfunction
```

Now, `echo` the return value of the function so we can see it in the following screenshot:

```
:echo PrepareIngredient2('sausage')
```

Figure 8.11 – Output of :echo PrepareIngredient('sausage')

Classes (Vim9script)

Legacy Vimscript classes work through odd syntax and generous application of duct tape – a limitation addressed by Vim9script. If you're interested in using classes, I recommend using Vim9script syntax, either by starting your script with `vim9script` or prefixing a command with `vim9cmd`.

If you need to use legacy Vimscript, I've included a legacy Vimscript section. It's clunkier but offers backward compatibility.

> **Under construction**
>
> Classes are a new feature in Vim9script, and the behaviors might change. See `:help vim9class` for information relevant to your Vim version.

Classes in Vim9script function just like classes do in other common languages. You'd define a class as follows:

```
class Dish
  this.ingredient: string
  this.dish_name: string

  def PrepareIngredient(has_spam: bool)
    this.dish_name = has_spam ? this.ingredient ..
        \ ' and spam' : this.ingredient
  enddef
endclass
```

You can create an object from the class using the `new()` method:

```
var bacon = Dish.new('bacon')
bacon.PrepareIngredient(true)
echo bacon.dish_name
```

You can see that the `dish_name` for the `bacon` object was changed to `bacon and spam`:

```
vim9script

class Dish
  this.ingredient: string
  this.dish_name: string

  def PrepareIngredient(has_spam: bool)
    this.dish_name = has_spam ? this.ingredient .. ' and spam' : this.ingredient
  enddef
endclass

var bacon = Dish.new('bacon')
bacon.PrepareIngredient(true)
echo bacon.dish_name
08_classes.vim                                    1,1           All
bacon and spam
```

Figure 8.12 – Classes in Vim9script

Classes in legacy Vimscript

While legacy Vimscript doesn't explicitly contain classes, dictionaries support having methods on them, supporting the object-oriented programming paradigm. It's clunky, but here's how you'd do it:

```
function PrepareIngredient(has_spam) dict
  let self.dish_name = a:has_spam == 1 ? self.ingredient . ' and spam'
      \ : self.ingredient
endfunction

let dish = {
  \ 'ingredient': 'sausage',
  \ 'dish_name': '',
  \ 'PrepareIngredient': function('PrepareIngredient')
  \ }
```

We define the function before defining a dictionary, and the `dict` keyword allows us to access dictionary attributes via self (just like in Python). Defining a `dict` effectively functions as class initialization.

Let's run it:

```
call dish.PrepareIngredient(1)
echo dish.dish_name
```

You can see the `dish_name` for the `dish` object is now set to `sausage and spam`:

```
function PrepareIngredient(has_spam) dict
  let self.dish_name = a:has_spam == 1 ? self.ingredient . ' and spam'
      \ : self.ingredient
endfunction

let dish = {
  \ 'ingredient': 'sausage',
  \ 'dish_name': '',
  \ 'PrepareIngredient': function('PrepareIngredient')
  \ }

call dish.PrepareIngredient(1)
echo dish.dish_name
../tutorial/08_classes.vim                                  1,1           All
sausage and spam
```

Figure 8.13 – Hacking together classes in legacy Vimscript

Understandably, these aren't fully-fledged classes, but they can be a useful tool as you're working with legacy Vimscript.

Lambda expressions

Lambdas are anonymous functions that can be really useful when working with somewhat straightforward logic.

Here's how we would define `PrepareIngredient` from the previous example using lambda expressions:

```
let PrepareIngredient = {ingredient -> ingredient . ' and spam'}
```

> **Syntax difference in Vim9script**
>
> In Vimscript 9, lambda expressions use a different syntax: `var lambda = (arg): type => expression`, trading `->` for `=>`, removing curly braces, and introducing parentheses around the argument.

Let's test it:

```
:echo PrepareIngredient('sausage')
```

You can see the output in the following screenshot:

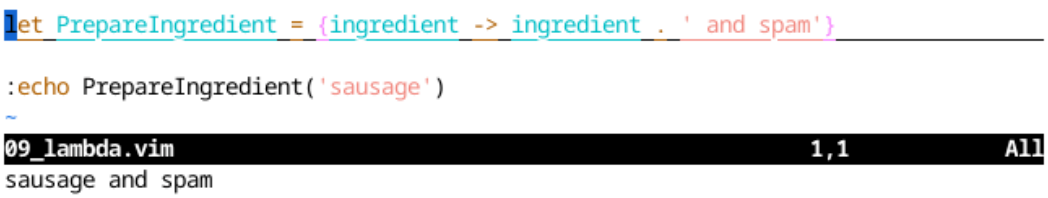

Figure 8.14 – Lambda expressions in Vimscript

Lambdas provide short and sweet syntax for writing compact functions.

Map and filter

Vimscript supports `map` and `filter`—higher-order functions (functions that operate on functions). Both functions take either a list or a dictionary as a first argument and a function as a second.

Say we want to remove every meal from a list that doesn't have spam in it. `HasSpam` will return `1` (true) if the dish contains spam, and `0` (false) otherwise, and we'll use the `filter` function to apply `HasSpam` to every element of the `dishes` list:

```
let dishes = ['spam omelet', 'sausage', 'bacon and spam']

function HasSpam(dish)
  if stridx(a:dish, 'spam') > -1
    return 1
  endif
  return 0
endfunction

call filter(dishes, 'HasSpam(v:val)')
echo dishes
```

What's stridx?

`stridx` is effectively a "contains" function. It returns an index of a substring within a string, and returns `-1` if the substring is not found within a string. See `:help stridx`.

Here's the output, as expected:

```
let dishes = ['spam omelet', 'sausage', 'bacon and spam']

function HasSpam(dish)
  if stridx(a:dish, 'spam') > -1
    return 1
  endif
  return 0
endfunction

call filter(dishes, 'HasSpam(v:val)')
echo dishes
10_map_and_filter.vim                                    1,1           All
['spam omelet', 'bacon and spam']
```

Figure 8.15 – A filter function in Vimscript

If you're coming from other programming languages, this syntax probably feels somewhat awkward. The second argument to the filter function is a string, which gets evaluated for every key-value pair of the dictionary.

Somewhat confusingly, when operating on lists, `v:key` refers to an item index, and `v:val` refers to the item value. When operating on dictionaries, more expectedly, `v:val` will get expanded to the dictionary value (while `v:key` could be used to access the key).

The `map` function behaves in a similar manner. It lets you modify each list item or dictionary value.

For example, let's add spam to each list item missing spam.

In this exercise, we'll reuse the `HasSpam` function from the earlier example. Lambdas come in especially useful with filter and map functions:

```
let dishes = ['spam omelet', 'sausage', 'bacon']

function HasSpam(dish)
  if stridx(a:dish, 'spam') > -1
    return 1
  endif
  return 0
endfunction

call map(dishes, 'HasSpam(v:val) ? v:val : v:val . '' and spam ''')
```

Verify that the results are as expected:

```
echo dishes
```

```
let dishes = ['spam omelet', 'sausage', 'bacon']

function HasSpam(dish)
  if stridx(a:dish, 'spam') > -1
    return 1
  endif
  return 0
endfunction

call map(dishes, 'HasSpam(v:val) ? v:val : v:val . '' and spam ''')
echo dishes
10_map_and_filter.vim                                    17,1          Bot
['spam omelet', 'sausage and spam ', 'bacon and spam ']
```

Figure 8.16 – Example of a map function in legacy Vimscript

Interacting with Vim

The `execute` command lets you parse and execute a string as a Vim command. For example, the two following statements will produce equivalent results:

```
echo dish . ' probably got spam in it'
execute 'echo dish ''probably got spam in it'''
```

> **Note**
>
> Quotes around are needed as otherwise this command would `:echo` the contents of each variable. If those variables are not defined, an error occurs.

You can use `normal` to execute keys just like the user would in normal mode. For instance, if you wanted to search for a word, `egg`, and delete it, you could do this:

```
normal /egg<cr>dw
```

> **Note**
>
> `<cr>` here needs to be typed with *Ctrl* + *v*, followed by the *Enter* key. However, `execute "normal /egg<cr>dw"` would use the literal string `<cr>` to represent the *Enter* key press. Just a quirk to be aware of.

Running normal like this will respect the user's mappings, so if you want to ignore custom mappings, you could use `normal!`:

```
normal! /egg<cr>dw
```

As a general recommendation, one should always use `:normal!` (with bang attribute) to be independent of what a user may have mapped the key to.

Another command can suppress output from other commands (such as `execute`): `silent`. Neither of these will produce any output:

```
silent echo dish . ' probably got spam in it'

silent execute 'echo dish ''probably got spam in it'''
```

Furthermore, `silent` can suppress the output from external commands and ignore prompts:

```
silent !echo 'this is running in a shell'
```

You can also check whether the Vim you're running in has a particular feature enabled:

```
if has('python3')
  echom 'Your Vim was compiled with Python 3 support!'
endif
```

You can view the full list of features via `:help feature-list`, but something worth noting is OS indicators: `win32`/`win64`, `macunix`/`osxdarwin` (macOS), or `unix`. These are extremely helpful if you're planning to build a cross-platform script.

File-related commands

Since Vim is a text editor, much of what you do operates on files. Vim provides a number of file-related functions.

You can manipulate file path information using `expand`:

```
echom 'Current file extension is ' . expand('%:e')
```

When passed a filename (through `%`, `#`, or shortcuts such as `<cfile>`), `expand` lets you parse the path using these modifiers:

- `:p` expand to full path
- `:h` head (last path component removed)
- `:t` tail (last path component only)
- `:r` root (one extension removed)
- `:e` extension only

See `:help expand()` for more information about these.

You can check that the file exists (aka can be read) by using `filereadable`:

```
if filereadable(expand('%'))
  echom 'Current file (' . expand('%:t') . ') is readable!'
endif
```

When executed from `12_file.vim`, the output would be as follows:

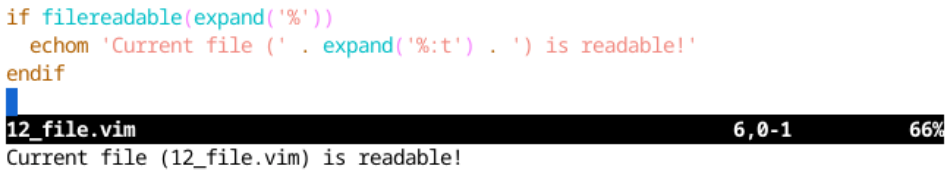

Figure 8.17 – Checking whether the current file is readable in Vimscript

Similarly, you can check you have write permissions to the file using `filewritable`.

You can perform the rest of the file operations using the `execute` command. For example, you'd write the following to open the `welcome.py` file:

```
execute 'edit welcome.py'
```

Prompts

There are two primary ways to prompt the user for input. You can either use `confirm` to display a multiple-choice dialog (such as **yes**/**no**/**cancel**), or `input` to process a more complex input.

The `confirm` function prompts the user with a dialog and multiple answers a user can select from. Let's try a simple example:

```
let answer = confirm('Add spam to a dish?', "&yes\n&no")
echo answer
```

If you execute the script, you'll get the following prompt:

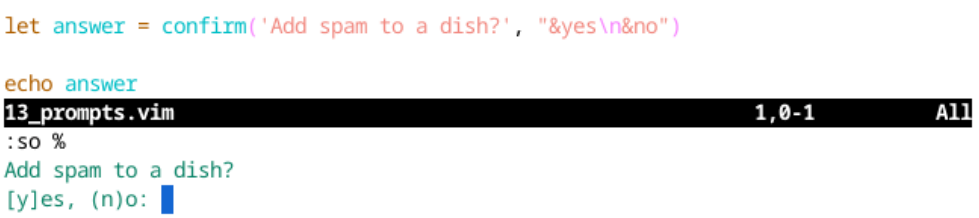

Figure 8.18 – Using Vimscript to show a prompt

Hitting *y* or *n* will select an option. Let's hit *y*:

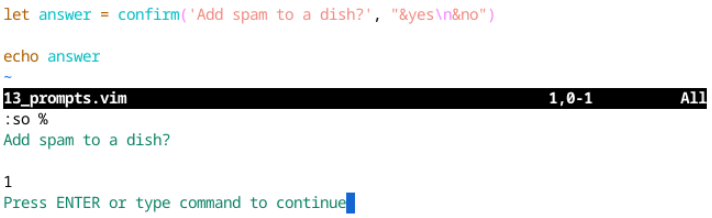

Figure 8.19 – Selecting the first option returns 1

The result of this is **1**. Now, what if we replay it and choose `no`?

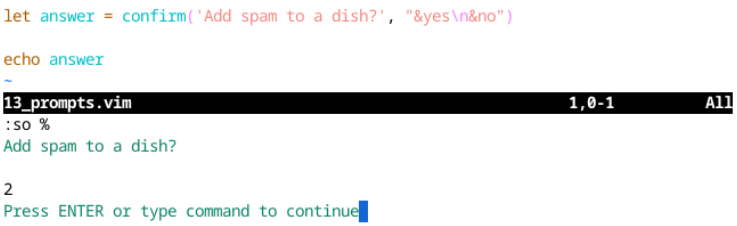

Figure 8.20 – Selecting the second option returns 2

We get **2**. As you can see, `confirm` returns an integer with the number of the selected choice.

Oh, and if you're running from a GUI, you'll get a dialog window pop up:

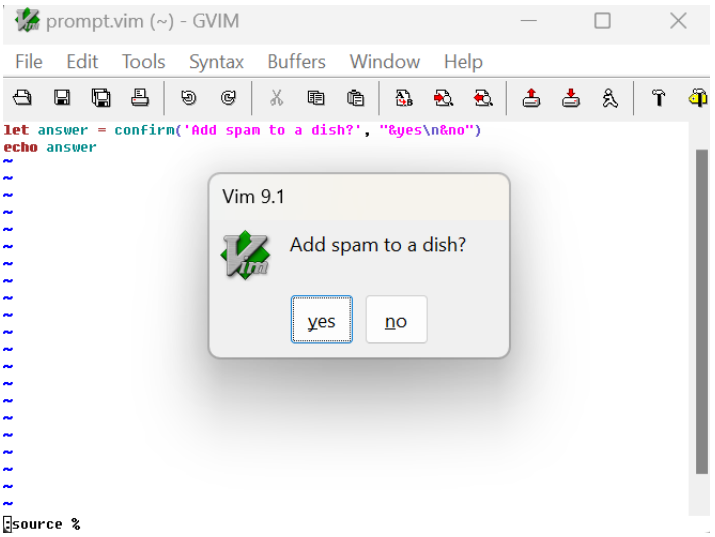

Figure 8.21 – A visual prompt in GVim

Now, let's get back to our original example:

```
let answer = confirm('Add spam to a dish?', "&yes\n&no")
echo answer
```

Here, you can see that `confirm` takes two arguments: a prompt to be displayed and a newline-separated (`\n`) list of options to select from. In the previous example, an option string is non-literal, since we want the newlines to be processed.

A set of ampersand (`&`) symbols is used to denote the letters representing each option (in the previous example, `y` and `n` become the available options). Here's another example:

```
let answer = confirm(
    \ 'Add spam to a dish?', "absolutely &yes\nhell &no")
```

This would display the following prompt:

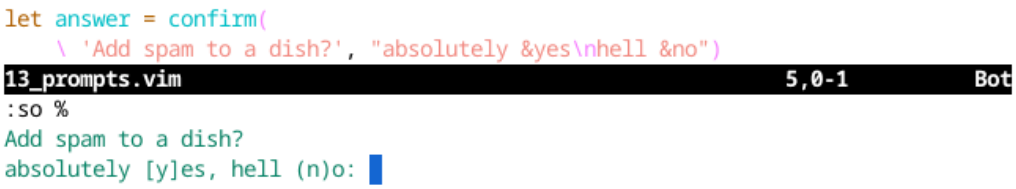

Figure 8.22 – You can specify which letter represents each option

Note that `y` and `n` are still the letters a user can press to reply to the prompt.

Lastly, `input` lets you work with free-form text input. Its use is fairly straightforward:

```
let ingredient = input('Please input an ingredient: ')
echo "\n"
echo 'We now serve ' . ingredient . ' and spam!'
```

> **Note**
>
> `echo "\n"` prints a newline, as otherwise your input and the next line will not be separated by a newline.

And this is how the prompt looks when executed:

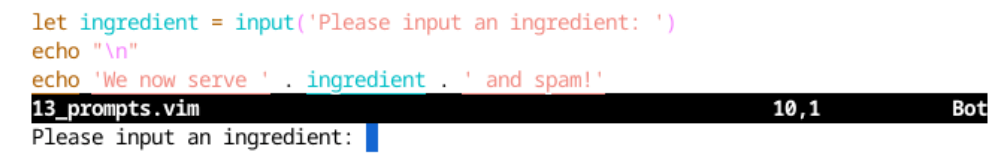

Figure 8.23 – A freeform text prompt

And this is what it looks like after we enter our string:

```
let ingredient = input('Please input an ingredient: ')
echo "\n"
echo 'We now serve ' . ingredient . ' and spam!'
13_prompts.vim                                    10,1          Bot
Please input an ingredient: egg
We now serve egg and spam!
Press ENTER or type command to continue
```

Figure 8.24 – A freeform text prompt output

However, a word of warning. If you're using `input` from inside a mapping, you must prefix it with `inputsave()` and follow it with `inputrestore()`. Otherwise, the rest of the characters in a mapping will be consumed into input. In fact, you should always use `inputsave()` and `inputrestore()` in case your function is ever used in a mapping. Here's an example of how to use them:

```
function InputIngredient()
  call inputsave()
  let ingredient = input('Please input an ingredient: ')
  call inputrestore()
  return ingredient
endfunction

nnoremap <leader>a = :let ingredient = InputIngredient()<cr>:echo
ingredient<cr>
```

Using :help

Most of the information about Vimscript is in Vim's `eval.txt`, which you can access by searching `:help eval`. Information about Vim9script is available in `vim9.txt` and `vim9sclass.txt`. Give it a read if you're ever feeling stuck or would like to learn more.

A word about style guides

Consistent style is important. One of the more prominent style guides for Vim is the one published by Google: `https://google.github.io/styleguide/vimscriptguide.xml`. It highlights some common development practices and outlines common pitfalls.

Here are some excerpts from the Google Vimscript style guide:

- Use two spaces for indents
- Do not use tabs
- Use spaces around operators

- Restrict lines to 80 columns wide
- Indent continued lines by four spaces
- Use `plugin-names-like-this`
- Use `FunctionNamesLikeThis`
- Use `CommandNamesLikeThis`
- Use `augroup_names_like_this`
- Use `variable_names_like_this`
- Always prefix variables with their scope
- When in doubt, apply Python style guide rules

Give the Google Vimscript style guide a read; it's rather useful even if you never plan on doing more than customizing your `.vimrc` file. It'll help with self-consistency.

Let's build a plugin

Let's try building a simple plugin; this way, we can learn by example.

A common task you have to perform when working with code is commenting out chunks of code. Let's build a plugin that does just that. Let's (uninspiringly) name our plugin `vim-commenter`.

Plugin layout

Since the Vim 8 release, there's thankfully only one way of structuring your plugins (which is also compatible with major plugin managers, such as vim-plug, Vundle, or Pathogen). The plugins are expected to have the following directory structure:

- `autoload/` lets you lazy load bits of your plugin (more on that later)
- `colors/` color schemes
- `compiler/` (language-specific) compiler-related functionality
- `doc/` documentation
- `ftdetect/` (file type-specific) file type detection settings
- `ftplugin/` (file type-specific) file type-related plugin code
- `indent/` (file type-specific) indentation-related settings
- `plugin/` the core functionality of your plugin
- `syntax/` (language-specific) defines language syntax group

As we develop our plugin, let's use Vim 8's new plugin functionality and place our plugin directory into `.vim/pack/plugins/start`. Since we decided to name our plugin commenter, we'll plop it into `.vim/pack/plugins/start/vim-commenter`.

Remember, the `plugins/` directory can have any name. See *Chapter 3, Follow the Leader – Plugin Management* for more information. The `start/` directory means that the plugin will be loaded on Vim startup.

Let's create a directory for it now:

```
$ mkdir -p ~/.vim/pack/plugins/start/vim-commenter
```

The basics

Let's start simple: let's get our plugin to add a key binding that comments out the current line by prefixing it with a Python-style comment (#).

Let's start in `~/.vim/pack/plugins/start/vim-commenter/plugin/commenter.vim`:

```
" Comment out the current line in Python.
function! commenter#Comment()
  let l:line = getline('.')
  call setline('.', '# ' . l:line)
endfunction

nnoremap gc :call commenter#Comment()<cr>
```

In the previous example, we've created a function that inserts # in front of the current line (`.`) and maps it to `gc`. As you might remember, `g`, while having some mappings assigned to it (see `:help g`), is effectively a free namespace for the user to fill, and `c` stands for "comment."

Save the file and load it (either using `:source` or by restarting Vim). Let's open a Python file and navigate to some line we'd like to comment:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
```

Figure 8.25 – An unsuspecting Python file

Now, try running `gc`:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
#     has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
:call commenter#Comment()
```

Figure 8.26 – Success-ish: a line has been commented out

Success! Well, kind of. First, the comment begins at the beginning of the line and not at the current indentation level, as the user might want. Also, the cursor hasn't moved from the current position, which might be a little annoying for the user. Let's fix these two issues.

You can get the indentation level of the line (in spaces) using the `indent` function:

```
let s:comment_string = '# '

" Comment out the current line in Python.
function! Comment()
  let l:i = indent('.') " Number of spaces.
  let l:line = getline('.')
  let l:cur_row = getcurpos()[1]
  let l:cur_col = getcurpos()[2]
  call setline('.', l:line[:l:i - 1] . s:comment_string .
      \ l:line[l:i:])
  call cursor(l:cur_row, l:cur_col + len(s:comment_string))
endfunction

nnoremap gc :call Comment()<cr>
```

Let's go back to our file:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
```

Figure 8.27 – An unsuspecting Python file

Now, run `gc` to comment out an indented line:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    # has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
:call Comment()
```

Figure 8.28 – Success: a line properly commented out!

Wonderful! But now, we'll probably need a way to uncomment the line as well! Let's change our function to `ToggleComment()`:

```
let s:comment_string = '# '

" Comment out the current line in Python.
function! ToggleComment()
  let l:i = indent('.') " Number of spaces.
  let l:line = getline('.')
  let l:cur_row = getcurpos()[1]
  let l:cur_col = getcurpos()[2]
  if l:line[l:i:l:i + len(s:comment_string) - 1] == s:comment_string
    call setline('.', l:line[:l:i - 1] .
```

```
        \ l:line[l:i + len(s:comment_string):])
    let l:cur_offset = -len(s:comment_string)
  else
    call setline('.', l:line[:l:i - 1] . s:comment_string .
        \ l:line[l:i:])
    let l:cur_offset = len(s:comment_string)
  endif
  call cursor(l:cur_row, l:cur_col + l:cur_offset)
endfunction

nnoremap gc :call ToggleComment()<cr>
```

Let's give it a shot! Reload the script, and go back to our file:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
```

Figure 8.29 – An unsuspecting Python file

Hit `gc` to comment the line:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    # has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
:call ToggleComment()
```

Figure 8.30 – A line successfully commented out

And hit `gc` again to uncomment it:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
:call ToggleComment()
```

Figure 8.31 – A line successfully uncommented

Let's make sure we cover corner cases! Let's try to comment out the line without indentation. Move your cursor to an unindented line:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
```

Figure 8.32 – Place your cursor on an unindented line

Hit `gc` to run our function:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):# def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
:call ToggleComment()
```

Figure 8.33 – Our plugin doesn't seem to work on unindented lines!

Oh no! Looks like our script is not working well when there's no indentation. Let's fix it:

```
let s:comment_string = '# '

" Comment out the current line in Python.
function! ToggleComment()
  let l:i = indent('.') " Number of spaces.
  let l:line = getline('.')
  let l:cur_row = getcurpos()[1]
  let l:cur_col = getcurpos()[2]
  let l:prefix = l:i > 0 ? l:line[:l:i - 1] : '' " Handle 0 indent
  if l:line[l:i:l:i + len(s:comment_string) - 1] == s:comment_string
    call setline('.', l:prefix . l:line[l:i + len(s:comment_string):])
    let l:cur_offset = -len(s:comment_string)
  else
    call setline('.', l:prefix . s:comment_string . l:line[l:i:])
    let l:cur_offset = len(s:comment_string)
  endif
  call cursor(l:cur_row, l:cur_col + l:cur_offset)
endfunction

nnoremap gc :call ToggleComment()<cr>
```

Let's save, reload, and run the script using `gc`:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

# def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
:call ToggleComment()
```

Figure 8.34 – Commenting out an unindented line

And run `gc` again to test uncommenting:

```
from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

def prepare_ingredient(ingredient):
    has_spam = random.choice([True,  False])
    return ingredient.prepare(has_spam)

def main():
    print('Scene: A cafe. A man and his wife enter.')
:call ToggleComment()
```

Figure 8.35 – Uncommenting an unindented line

Wonderful! The very basic version of our plugin is complete!

Housekeeping

So far, we've had our plugin all within one file. Let's see how we can break it down into multiple files to keep our newly created project organized! Look at the list in the *Plugin layout* section of this chapter.

First, you can see that the `ftplugin/` directory contains file type-specific plugin configuration. Right now, most of our plugin is actually pretty independent from working with Python, except for the `s:comment_string` variable. Let's move it out to `<...>/vim-commenter/ftplugin/python.vim`:

```
" String representing inline Python comments.
let g:commenter#comment_str = '# '
```

We've changed the scope from `s:` to `g:` (since the variable is now used in different scripts) and added the `commenter#` namespace to avoid namespace collision.

The name should also be updated in `<...>/vim-commenter/plugin/commenter.vim`. Now might be a good time to test those substitution commands you learned earlier in this book:

```
:%s/\<s:comment_string\>/g:commenter#comment_str/g
```

Another directory of interest is `autoload/`. Currently, whenever Vim starts, it will parse and load `g:commenter#ToggleComment`. That's not very fast. Instead, we can choose to move the function

to the `autoload/` directory. The name of the file needs to correspond to its namespace; in this case, it's `commenter`. Let's create `<...>/vim-commenter/autoload/commenter.vim`:

```
" Comment out the current line in Python.
function! g:commenter#ToggleComment()
  let l:i = indent('.') " Number of spaces.
  let l:line = getline('.')
  let l:cur_row = getcurpos()[1]
  let l:cur_col = getcurpos()[2]
  let l:prefix = l:i > 0 ? l:line[:l:i - 1] : '' " Handle 0 indent
  if l:line[l:i:l:i + len(g:commenter#comment_str) - 1] ==
      \ g:commenter#comment_str
    call setline('.', l:prefix .
        \ l:line[l:i + len(g:commenter#comment_str):])
    let l:cur_offset = -len(g:commenter#comment_str)
  else
    call setline('.', l:prefix . g:commenter#comment_str .
        \ l:line[l:i:i])
    let l:cur_offset = len(g:commenter#comment_str)
  endif
  call cursor(l:cur_row, l:cur_col + l:cur_offset)
endfunction
```

At this point, the only thing left in `<...>/vim-commenter/plugin/commenter.vim` is the mapping:

```
nnoremap gc :call g:commenter#ToggleComment()<cr>
```

Here's how our plugin will get loaded when a user is working with Vim:

- User opens Vim, and `<...>/vim-commenter/plugin/commenter.vim` is loaded. Our `gc` mapping is now registered.
- User opens a Python file, and `<...>/vim-commenter/ftplugin/python.vim` is loaded. `g:commenter#comment_str` is initialized.
- User runs `gc`, which loads and executes `g:commenter#ToggleComment` within `<...>/vim-commenter/autoload/commenter.vim`.

One directory we haven't given much love to yet is `doc/`. Vim is known for having extensive documentation, and we have a recommendation to uphold this. Let's add `<...>/vim-commenter/doc/commenter.txt`:

```
*commenter.txt* Our first commenting plugin.
*commenter*
```

```
==============================================================================
CONTENTS *commenter-contents*

1. Intro.....................................|commenter-intro|
2. Usage.....................................|commenter-usage|

==============================================================================
1. Intro *commenter-intro*

Have you ever wanted to comment out a line with only three presses of
a button? Now you can! The new and wonderful vim-commenter lets you
comment out a single line in Python quickly!

2. Usage *commenter-usage*

This wonderful plugin supports the following key bindings:

gc: toggle comment on a current line

That's it for now. Thanks for reading!

vim:tw=78:ts=2:sts=2:sw=2:ft=help:norl:
```

Vim help has its own format, but here are some highlights:

- `*help-tag*` is used to denote a help tag. Whenever you run `:help help-tag`, Vim takes you to a file containing `*help-tag*`, and places the cursor right at the tag.
- `Text...|help-tag|` is used for navigation within a help file. It lets the reader jump to the desired tags from this section.
- All the `===` lines are just for pretty looks. They don't actually mean anything.
- A line such as `vim:tw=78:ts=2:sts=2:sw=2:ft=help:norl:` lets you tell Vim how to display a file when editing it (all of these are options you can set using the `set` keyword). This becomes really useful for files without clearly identifiable file types (such as `.txt` files).

You can learn more about Vim's help format by reading `:help help-writing`. The easiest thing, though, is to find some popular plugins and copy what they do.

Now, you'll be able to visit the entries you added to the help file:

```
:help commenter-intro
```

> **In case of trouble**
>
> If the help page doesn't show up, try getting Vim to manually index it by running `:helptags ~/.vim/pack/plugins/start/vim-commenter/doc`.

Here's a screenshot of Vim help taking you to the requested section:

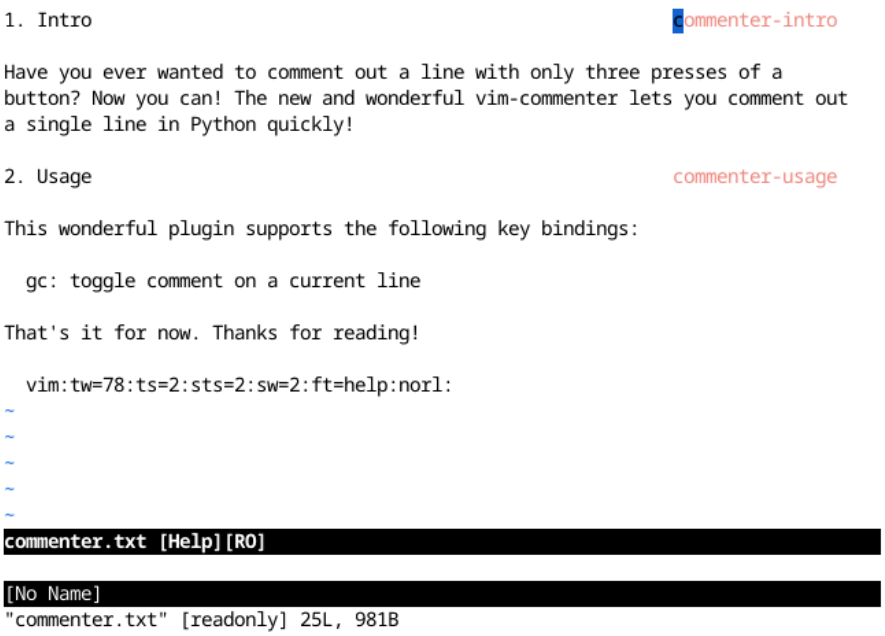

Figure 8.36 – The :help commenter-intro output

Improving our plugin

There are many paths we can take our plugin along, but let's focus on two main issues we have right now:

- Our plugin fails with spectacular errors when we try to execute it in any other language
- The plugin does not provide a way to operate on multiple lines at once

Let's start with the first problem: making the plugin work across different languages. Right now, if you try to execute the plugin in, for example, a `.vim` file, you'll get the following volley of errors:

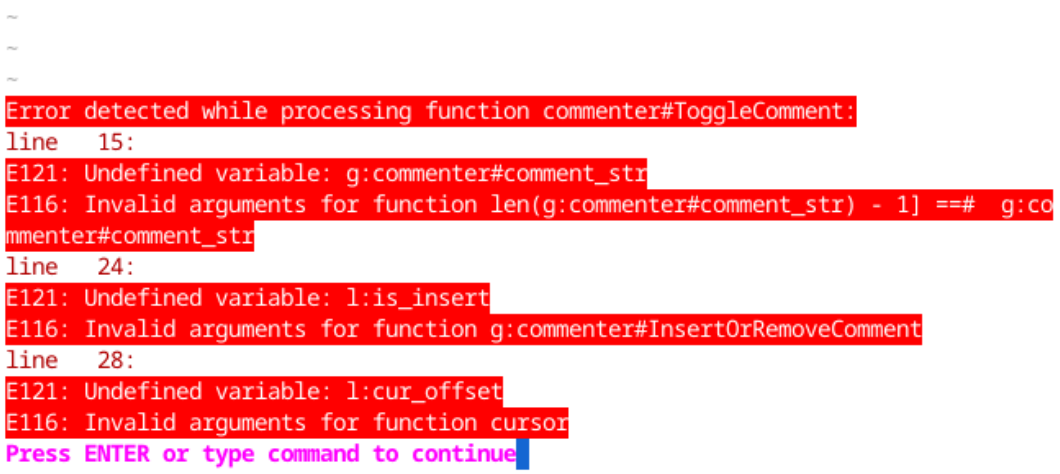

Figure 8.37 – An error when trying to run our plugin on a non-Python file

That's because we define `g:commenter#comment_str` in `<...>/vim-commenter/ftplugin/python.vim`, and the variable is only defined when we're working with a Python file.

Vim syntax files define what the comments look like for each language, but they're not very consistent, and the logic to parse those and all the corner cases is outside of the scope of this book.

However, we can at least get rid of the nasty error, and make our own!

The canonical way of checking whether a variable exists is with `exists`. Let's add a new function to `<...>/vim-commenter/autoload/commenter.vim`, which would throw a custom error if `g:commenter#comment_str` is not set:

```
" Returns 1 if g:commenter#comment_str exists.
function! g:commenter#HasCommentStr()
  if exists('g:commenter#comment_str')
    return 1
  endif
  echom "vim-commenter doesn't work for filetype " . &ft . " yet"
  return 0
endfunction

" Comment out the current line in Python.
function! g:commenter#ToggleComment()
  if !g:commenter#HasCommentStr()
```

```
    return
  endif
  let l:i = indent('.') " Number of spaces.
  let l:line = getline('.')
  let l:cur_row = getcurpos()[1]
  let l:cur_col = getcurpos()[2]
  let l:prefix = l:i > 0 ? l:line[:l:i - 1] : '' " Handle 0 indent
  if l:line[l:i:l:i + len(g:commenter#comment_str) - 1] ==#
      \ g:commenter#comment_str
    call setline('.', l:prefix .
        \ l:line[l:i + len(g:commenter#comment_str):])
    let l:cur_offset = -len(g:commenter#comment_str)
  else
    call setline('.', l:prefix . g:commenter#comment_str .
        \ l:line[l:i:])
    let l:cur_offset = len(g:commenter#comment_str)
  endif
  call cursor(l:cur_row, l:cur_col + l:cur_offset)
endfunction
```

Now, we get a message when we try to comment out a line in a non-Python file:

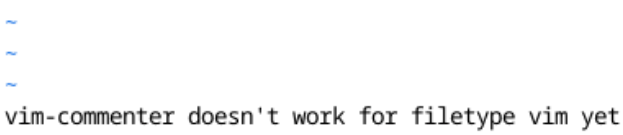

Figure 8.38 – A proper error message when running our plugin on a non-Python file

A much better user experience if you ask me.

And now, let's add a way for our plugin to be invoked on multiple lines. The easiest thing to do would be to allow the user to prefix our `gc` command with a number, and we'll do just that.

Vim lets you access a number that prefixes a mapping by using `v:count`. Even better, there's `v:count1`, which defaults to 1 if no count was given (this way, we can reuse more of our code).

Let's update our mapping in `<...>/vim-commenter/plugin/commenter.vim`:

```
nnoremap gc :<c-u>call g:commenter#ToggleComment(v:count1)<cr>
```

`<c-u>` is required to be used with `v:count` and `v:count1`. You can check `:help v:count` or `:help v:count1` for an explanation.

In fact, we can also add a visual mode mapping to support visual selection:

```
vnoremap gc :<cu>call g:commenter#ToggleComment(
    \ line("'>") - line("'<") + 1)<cr>
```

> **Clever way to count lines**
>
> `line("'>")` gets the line number of the end of the selection, while `line("'<")` gets the line number of the beginning of the selection. Subtract the beginning line number from the end, add one, and we have ourselves a line count!

Now, let's update `<...>/vim-commenter/autoload/commenter.vim` with a few new methods:

```
" Returns 1 if g:commenter#comment_str exists.
function! g:commenter#HasCommentStr()
  if exists('g:commenter#comment_str')
    return 1
  endif
  echom "vim-commenter doesn't work for filetype " . &ft . " yet"
  return 0
endfunction

" Detect smallest indentation for a range of lines.
function! g:commenter#DetectMinIndent(start, end)
  let l:min_indent = -1
  let l:i = a:start
  while l:i <= a:end
    if l:min_indent == -1 || indent(l:i) < l:min_indent
      let l:min_indent = indent(l:i)
    endif
    let l:i += 1
  endwhile
  return l:min_indent
endfunction

function! g:commenter#InsertOrRemoveComment(lnum, line, indent, is_
insert)
  " Handle 0 indent cases.
  let l:prefix = a:indent > 0 ? a:line[:a:indent - 1] : ''
  if a:is_insert
    call setline(a:lnum, l:prefix . g:commenter#comment_str .
        \ a:line[a:indent:])
  else
    call setline(
```

```
        \ a:lnum, l:prefix .
              \ a:line[a:indent + len(g:commenter#comment_str):]
  endif
endfunction

" Comment out the current line in Python.
function! g:commenter#ToggleComment(count)
  if !g:commenter#HasCommentStr()
    return
  endif
  let l:start = line('.')
  let l:end = l:start + a:count - 1
  if l:end > line('$') " Stop at the end of file.
    let l:end = line('$')
  endif
  let l:indent = g:commenter#DetectMinIndent(l:start, l:end)
  let l:lines = l:start == l:end ?
      \ [getline(l:start)] : getline(l:start, l:end)
  let l:cur_row = getcurpos()[1]
  let l:cur_col = getcurpos()[2]
  let l:lnum = l:start
  if l:lines[0][l:indent:l:indent +
      \ len(g:commenter#comment_str) - 1] ==#
      \ g:commenter#comment_str
    let l:is_insert = 0
    let l:cur_offset = -len(g:commenter#comment_str)
  else
    let l:is_insert = 1
    let l:cur_offset = len(g:commenter#comment_str)
  endif
  for l:line in l:lines
    call g:commenter#InsertOrRemoveComment(
        \ l:lnum, l:line, l:indent, l:is_insert)
    let l:lnum += 1
  endfor
  call cursor(l:cur_row, l:cur_col + l:cur_offset)
endfunction
```

This script is now much bigger, but it's not as scary as it looks! Here, we've added two new functions:

- `g:commenter#DetectMinIndent` finds the smallest indent within a given range. This way, we make sure to indent the outermost section of the code
- `g:commenter#InsertOrRemoveComment` either inserts or removes a comment within a given line and at a given indentation level

Let's test-run our plugin. Let's, say, run it with `11gc`:

```
#!/usr/bin/python

from kitchen import bacon, egg, sausage
import random

INGREDIENTS = [egg.Egg(), bacon.Bacon(), sausage.Sausage()]

# def prepare_ingredient(ingredient):
#     has_spam = random.choice([True,  False])
#     return ingredient.prepare(has_spam)
#
# def main():
#     print('Scene: A cafe. A man and his wife enter.')
#     print('Man: Well, what\'ve you got?')
#     menu = []
#     for ingredient in INGREDIENTS:
#         menu.append(prepare_ingredient(ingredient))
#     print('Waitress: Well, there\'s', ', '.join(menu))


if __name__ == '__main__':
    main()
~
:call g:commenter#ToggleComment(v:count1)
```

Figure 8.39 – Multiline commenting in all its glory

Ta-da! Now, our little plugin can comment out multiple lines! Give it a go with a few more tries to make sure we covered corner cases such as commenting in the visual mode, going past the end of the file, commenting and uncommenting a single line, and so on.

Distributing the plugin

Effectively, we're all set up to distribute the plugin. Just a few things left.

Update the documentation, and add a `README.md` file to let people know what your plugin does (this can be copied from the intro of your plugin). You'll also want to add a `LICENSE` file, indicating the license under which you're distributing the plugin. You can distribute the plugin under the same license as Vim (:help license), or choose your own (GitHub has a helpful link for this purpose: `https://choosealicense.org`).

Now, you'll just have to turn `$HOME/.vim/pack/plugins/vim-commenter` into a Git repository, and upload it somewhere.

At the time of writing this book, GitHub is the go-to bastion of freedom for storing code (however, as SourceForge proved around 2015, times change).

Let's give it a shot:

```
$ cd $HOME/.vim/pack/plugins/start/vim-commenter
$ git init
$ git add .
$ git commit -m "First version of the plugin is ready!"
$ git remote add origin <repository URL>
$ git push origin master
```

Done! You're now ready to distribute the plugin, and plugin managers such as vim-plug can now pick up your plugin!

Where to take the plugin from here

There's a lot more room for improvement, but we'll take a break here. You're welcome to continue working on this plugin on your own—you can add visual selection support, make it work with additional languages, or do whatever it is you would like with it.

Further reading

Vimscript is a long and complex topic, and this chapter only brushes it. If you want to learn more, there are a few options.

You can read `:help eval`, which contains most of the information about Vimscript, as well as `:help vim9`, `:help usr_51.txt`, and `:help usr_52.txt` (which cover writing plugins) to learn more about Vim9script.

You can also choose to follow a tutorial online or pick up a book. A lot of people recommend *Learn Vimscript the Hard Way* by Steve Losh. It's available online at `http://learnvimscriptthehardway.stevelosh.com/` (you can buy a paper copy from the website as well).

Summary

That was quite a bit of work! Let's do a quick recap!

We've learned that Vimscript lets you take Vim anywhere you want, limiting your productivity only by your imagination. We've covered setting and manipulating variables, working with lists and dictionaries, surfacing output, and control flows using `if`, `for`, and `while` statements. We've also covered functions, lambda expressions, the Vimscript equivalent of classes, as well as some more functional approaches using map and filter functions. We've also looked at Vim-specific commands and functions.

We've also built our first plugin, vim-commenter. The plugin lets you comment and uncomment lines in a Python file at the press of a button (well, two). We've learned how to structure our plugins, and how to use Vimscript to accomplish our goals. We've even brushed on distributing the plugin!

Lastly, we covered a few possible directions you can take for learning Vimscript, including digging into `:help eval` or picking a book up off the (possibly virtual) shelf.

In the next chapter, we'll take a look at Neovim, a developer effort that tries to build and improve upon Vim.

9
Where to Go from Here

Welcome to the last chapter of mastering Vim. You have now begun your journey into the wonderful world of Vim.

This book concludes with a few final thoughts:

- Healthy text editing habits, pulled from Bram Moolenaar's presentation
- Taking modal interfaces outside of Vim and into other IDEs, web browsers, and everywhere else
- Some of the Vim communities and recommended reading

Seven habits of effective text editing

This section is a condensed version of Bram Moolenaar's article from 2000 and a subsequent presentation from 2007. It's good; give it a read on Bram's website: `https://moolenaar.net/habits.html`. In case you decide to skip reading the whole thing, there follows a very high-level summary.

Since developers spend so much time reading and editing code, Bram highlights an important cycle when it comes to improving your text-editing experience:

1. Detect inefficiency.
2. Find a quicker way.
3. Make it a habit.

Those three steps are augmented with numerous examples. Here's one of the examples for each:

1. **Detect inefficiency**: Moving around takes a lot of time.
2. **Find a quicker way**: Often, you're looking for something that's already there. You can search for a piece of text to move faster. Or you can take a step or two further:
    - Use `*` to search for a word under the cursor
    - Use `:set incsearch` to search as you type
    - Use `:set hlsearch` to highlight every instance of a search pattern on the screen

3. **Make it a habit**: Use what you've learned! Set `incsearch` and `hlsearch` in your `.vimrc` file. Use `*` every time you catch yourself using the `/` command to search for a pattern near your cursor.

Modal interfaces everywhere

You've read through this book, and now, hopefully, you think that modal interfaces are pretty great. How can you get more of that?

Many applications support some sort of modal interactions with them, particularly Vi-friendly ones.

Many mature text editors and IDEs provide Vi-like key bindings for moving around and manipulating text. Here are a few of them (with the corresponding URLs):

- Evil is a Vi layer for Emacs: `https://github.com/emacs-evil/evil`
- IdeaVim is a Vim emulator for IDEA-based IDEs (IntelliJ IDEA, PyCharm, CLion, PhpStorm, WebStorm, RubyMine, AppCode, DataGrip, GoLand, Cursive, and Android Studio): `https://github.com/JetBrains/ideavim`
- Eclim lets you access Eclipse features from Vim: `http://eclim.org`
- Vrapper adds Vi-like key bindings to Eclipse: `http://vrapper.sourceforge.net/home`

There are many others, so if other editors accomplish your tasks better than Vim (because you might be locked into a particular IDE) but you enjoy using Vim key bindings, then this might be an approach for you.

A Vim-like web browsing experience

The modern developer workflow is very web-heavy, and I have my browser open most of the time when I'm working on code. Sometimes, I feel like switching my focus from keyboard-driven workflow in Vim to mouse-driven web browsing, which detracts from my sense of flow. To avoid that, I use add-ons that enable Vi-like key bindings in the browser.

It's hard to predict the future landscape when it comes to web browsers, but this section is based on the most popular browsers as of now.

Vimium and Vimium-FF

Vimium is a Chrome extension that enhances web browsing by allowing you to use Vim-friendly key bindings to navigate pages. It has also been ported to Firefox under the name **Vimium-FF**.

Vimium is available from the Chrome Web Store or at `https://vimium.github.io`. **Vimium-FF** is available from `https://addons.mozilla.org/en-US/firefox/addon/vimium-ff`.

For instance, if you hit *f*, **Vimium** will highlight every link on a page with a letter or a combination of letters (similar to the way `EasyMotion` works in Vim). This is shown in the following screenshot:

Figure 9.1 – Vimium navigation in Chrome

Pressing the right combination of letters will open a link or place your cursor in a textbox. **Vimium** supports visual selection to copy text without using a mouse: hitting *v* once enters a caret mode (where you can move the cursor around the page), and hitting *v* again enters a visual selection mode. Most movement keys familiar to you from Vim work in these modes, as shown in the following screenshot:

Figure 9.2 – Visual mode selection in Vimium

Once you select the text in a visual mode, press *y* to copy (yank) it.

Vimium also provides an omnibar for switching between tabs (*T*), opening URLs/history entries (*o*/*O*), and bookmarks (*b*/*B*), as shown in the following screenshot:

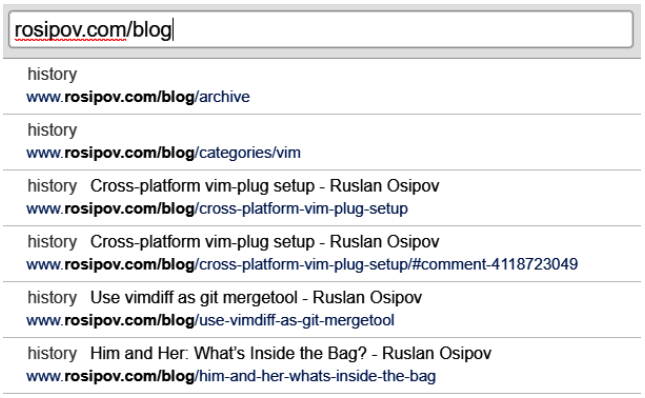

Figure 9.3 – Vimium's omnibar

Finally, help is available at the press of a button. Press *?* to open a help page and learn more about **Vimium** features, as shown in the following screenshot:

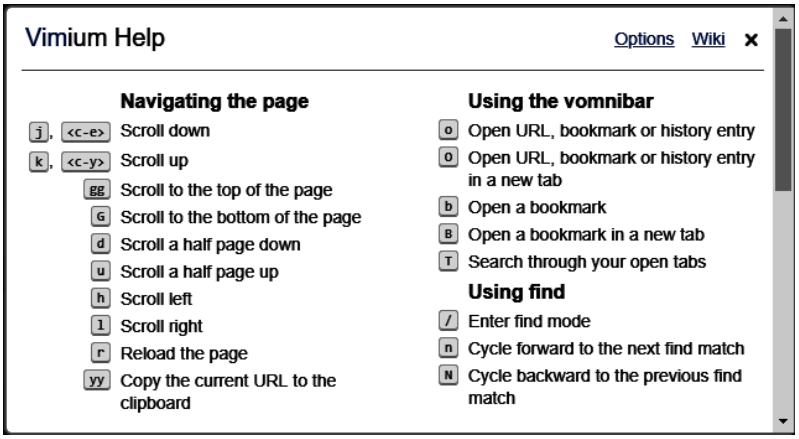

Figure 9.4 – Vimium help page, invoked by pressing ?

Alternatives

Vimium and **Vimium-FF** are possibly the most popular extensions as of the moment of writing (based on the number of users on the Chrome Web Store and on the Firefox Add-Ons website). There are many more available, and most mature browsers have Vi-like plugins. Here are a few more examples:

- Google Chrome also has extensions, including cVim or Vrome, that perform a similar function to **Vimium**, each providing slightly different functionality. Extensions such as **wasavi** focus on using Vim emulators to edit text areas.

- Safari supports **Vimari**, a port of **Vimium**.
- Mozilla Firefox has more add-ons similar to **Vimium-FF**: **Vim Vixen** and **Tridactyl** to name a few.
- Opera supports the installation of Chrome extensions.

Vim everywhere else

There are solutions for editing text in Vim in every text field on your system! In particular, there is **vim-anywhere** on Linux and macOS, and **Text Editor Anywhere** on Windows.

vim-anywhere for Linux and macOS

vim-anywhere lets you invoke **gVim** to edit any text on your Linux or macOS machine. vim-anywhere is available from `https://github.com/cknadler/vim-anywhere`. Once installed, place your cursor in a text field, and press *Ctrl* + *Cmd* + *v* on macOS or *Ctrl* + *Alt* + *v* on Linux. Depending on your platform, **vim-anywhere** will open either **MacVim** or **gVim**, as shown in the following screenshot:

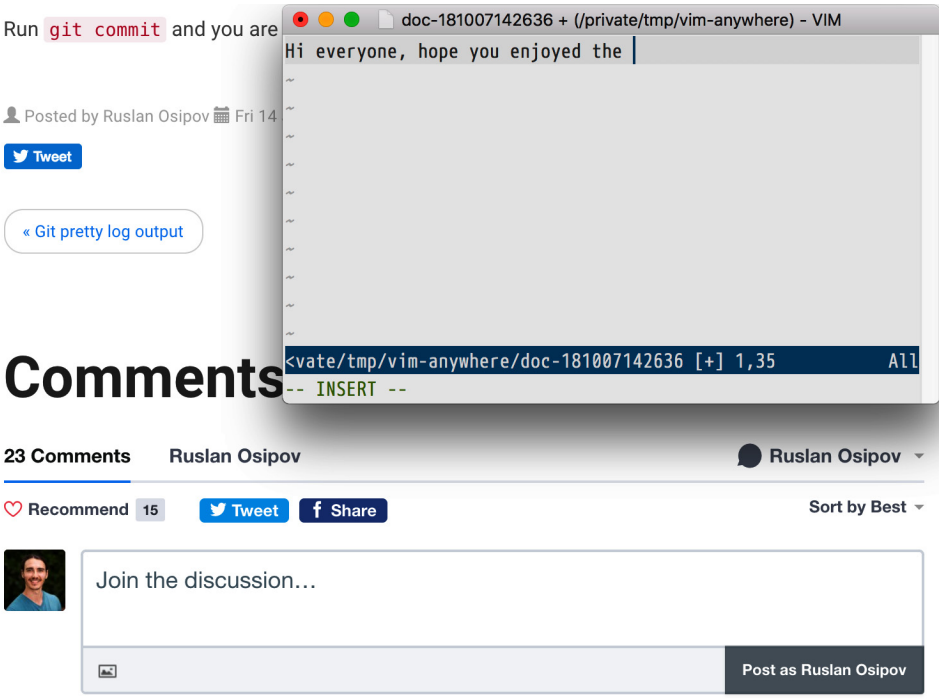

Figure 9.5 – vim-anywhere running on macOS

Save the buffer, exit **MacVim** or **gVim**, and vim-anywhere will insert the buffer contents into the original text field.

Text Editor Anywhere for Windows

Text Editor Anywhere allows you to select any text, open it in an editor of your choice, and insert the modified text back once you're done editing. **Text Editor Anywhere** is available for download from `https://www.portablefreeware.com/index.php?id=2188`.

I use Text Editor Anywhere whenever I work with Windows, having it configured to open **gVim** on a selected text when *Alt* + *a* is pressed.

The following screenshot shows what it looks like:

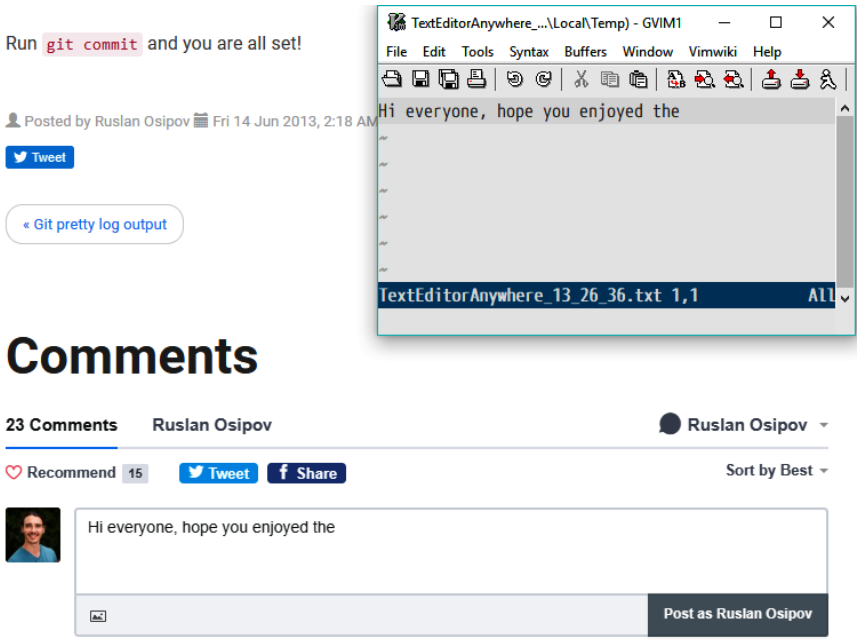

Figure 9.6 – Text Editor Anywhere running on Windows

When I'm done writing, I save the buffer and quit **gVim**. **Text Editor Anywhere** populates the text area with the saved buffer contents.

Neovim

Neovim aims to make Vim easier to maintain for its core developers, as well as make plugin development and various integrations easier. We'll look at the following:

- Why does Neovim matter?
- How to install and configure Neovim

- Synchronizing Vim and Neovim configuration
- Neovim-specific plugins

I'm not sure if this chapter will make you more productive, but I think Neovim is great for the Vim community and introduces some interesting ideas. Enjoy!

Why make another Vim?

Neovim is a fork of Vim that branched out into its own thing in 2014. Neovim aims to address a few core issues about Vim:

- Working with a 30-year-old code base while maintaining backward compatibility is hard.
- It's difficult to write certain kinds of plugins, asynchronous operations being a huge culprit (asynchronous support has been added to Vim in version 8.0, sometime after Neovim was forked).
- In fact, writing plugins is difficult overall and requires the developer to be comfortable with Vimscript.
- Vim is difficult to use on modern systems without tinkering with `.vimrc`.

Neovim aims to solve these problems with the following methods:

- Large-scale refactoring of the Vim code base, including choosing a single style guide, increasing test coverage
- Removing support for legacy systems
- Shipping Neovim with modern defaults
- Providing a rich API for plugins and external programs to talk to, including Python and Lua plugin support

Vim is installed on many machines, which makes backward compatibility and rare corner cases important. By branching out, Neovim is able to move faster, experiment, make mistakes, and make Vim even better than it currently is.

Neovim matters because it makes it easier to add new features as time goes on and develop plugins. Hopefully, it'll attract more developers and bring more perspectives and fresh ideas to the table as time goes on.

Installing and configuring Neovim

> **Important note**
>
> Neovim and its installation instructions are available from GitHub at `https://github.com/neovim/neovim`. You can either download the binary or install it through one of the package managers. The installation instructions are rather detailed and may change rather quickly, so you should give them a read at `https://github.com/neovim/neovim/wiki/Installing-Neovim`.
>
> If you are working on a Debian-based Linux distribution, you can install Neovim by running `$ sudo apt install neovim` and `$ python3 -m pip install neovim` to add Python3 to support neovim.

Once you install Neovim, it's available through the `nvim` command:

```
$ nvim
```

You're greeted by a screen similar to a vanilla Vim intro screen:

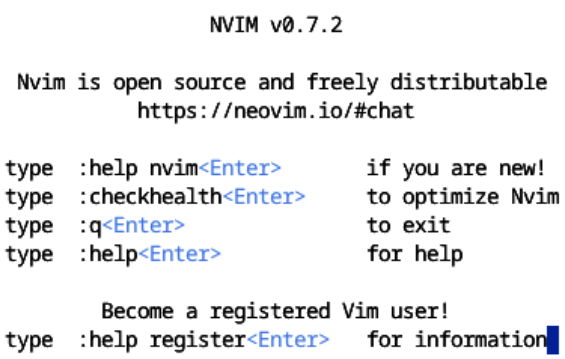

Figure 9.7 – Neovim intro screen

All of the commands familiar to you from Vim will work, and Neovim uses the same configuration format as Vim. However, your `.vimrc` file is not picked up automatically.

Neovim adheres to the XDG base directory specification, which suggests placing all of your configuration files into the `~/.config` directory. Neovim configuration is stored inside the `~/.config/nvim` directory:

- `~/.vimrc` becomes `~/.config/nvim/init.vim`
- `~/.vim/` becomes `~/.config/nvim/`

Most likely, you'll want to symlink your Neovim configuration to your Vim configuration:

```
$ mkdir -p $HOME/.config
$ ln -s $HOME/.vim $HOME/.config/nvim
$ ln -s $HOME/.vimrc $HOME/.config/nvim/init.vim
```

All done! Neovim will now read your `.vimrc` file!

Under Windows, Neovim configuration is likely located in `C:\Users\%USERNAME%\AppData\Local\nvim`.

You can configure Windows symlinks as follows:

```
$ mklink /D %USERPROFILE%\AppData\Local\nvim %USERPROFILE%\vimfiles
$ mklink %USERPROFILE%\AppData\Local\nvim\init.vim %USERPROFILE%\_
vimrc
```

Checking health

The intro screen suggests you run `:checkhealth`; let's give it a shot:

```
:checkhealth
```

You will be greeted by a screen that may look something like this:

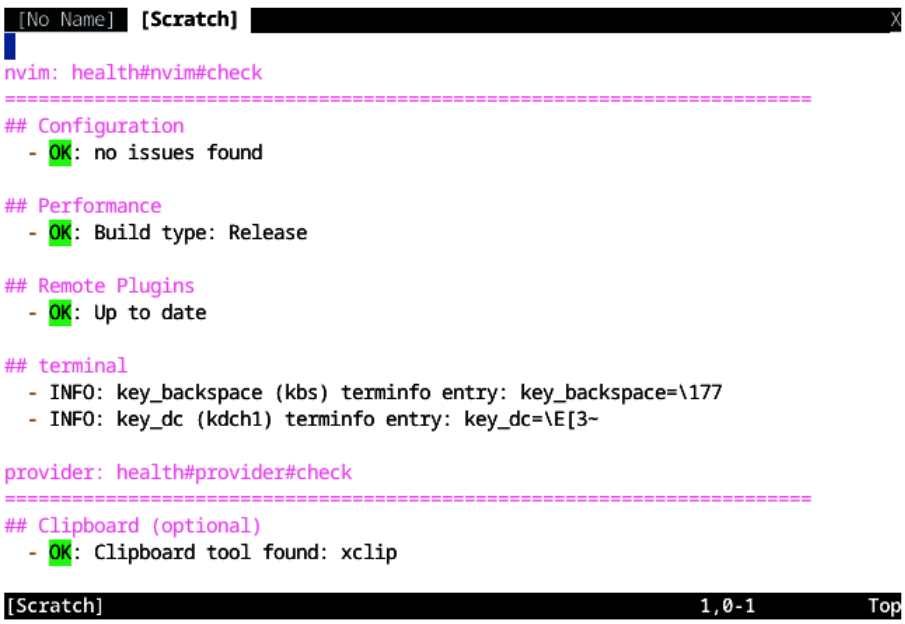

Figure 9.8 – Neovim :checkhealth output

Neovim health checks will outline everything wrong with your Neovim setup and suggest ways to fix those issues. You should go through the list and fix the errors relevant to you.

Sane defaults

Neovim comes with a number of different defaults than Vim. The defaults are meant to be more applicable to working with a text editor in the modern world. Compared to using Vim with an empty `.vimrc`, some of the more noticeable defaults include enabled syntax highlighting, sensible indentation settings, `wildmenu`, highlighted search results, and search-as-you-type.

You can check `:help nvim-defaults` from within Neovim to learn more about the defaults.

If you wanted to synchronize your settings between Vim and Neovim, you could add the following to your `~/.vimrc` (which is hopefully symlinked to `~/.config/nvim/init.vim`):

```
if !has('nvim')
 set nocompatible " not compatible with Vi
 filetype plugin indent on " mandatory for modern plugins
 syntax on " enable syntax highlighting
 set autoindent " copy indent from the previous line
 set autoread " reload from disk
 set backspace=indent,eol,start " modern backspace behavior
 set belloff=all " disable the bell
 set cscopeverbose " verbose cscope output
 set complete-=i " don't scan current on included
 " files for completion
 set display=lastline,msgsep " display more message text
 set encoding=utf-8 " set default encoding
 set fillchars=vert:|,fold: " separator characters
 set formatoptions=tcqj " more intuitive autoformatting
 set fsync " call fsync() for robust file saving
 set history=10000 " longest possible command history
 set hlsearch " highlight search results
 set incsearch " move cursor as you type when searching
 set langnoremap " helps avoid mappings breaking
 set laststatus=2 " always display a status line
 set listchars=tab:>\ ,trail:-,nbsp:+ " chars for :list
 set nrformats=bin,hex " <c-a> and <c-x> support
 set ruler " display current line # in a corner
 set sessionoptions-=options " do not carry options across sessions
 set shortmess=F " less verbose file info
 set showcmd " show last command in the status line
 set sidescroll=1 " smoother sideways scrolling
 set smarttab " tab setting aware <Tab> key
```

```
  set tabpagemax=50 " maximum number of tabs open by -p flag
  set tags=./tags;,tags " filenames to look for the tag command
  set ttimeoutlen=50 " ms to wait for next key in a sequence
  set ttyfast " indicates that our connection is fast
  set viminfo+=! " save global variables across sessions
  set wildmenu " enhanced command line completion
endif
```

I've left some short comments, attempting to briefly describe each one of these settings, and you can learn more about them by checking the corresponding `:help` entry.

Recommended reading and communities

This book doesn't aim to be a complete source of information about Vim, so there's a lot more to learn and explore. Depending on your preferred learning style, you might want to comb through the Vim manual by running `:help usr_toc.txt` (which can be read from beginning to the end), head to the community chat groups or mailing lists, or dive deeper into educational materials.

This section covers some possible routes you can take.

Mailing lists

Vim has a few primary mailing lists that you can browse or subscribe to. Details for each mailing list are listed at `https://www.vim.org/maillist.php`, but here are a few primary ones:

- `vim-announce@vim.org` is an official announcement channel; the archive is available at `https://groups.google.com/forum/#!forum/vim_announce`.
- `vim@vim.org` is the primary user support mailing list; the archive is available at `https://groups.google.com/forum/#!forum/vim_use`.
- `vim-dev@vim.org` is the mailing list used by Vim developers; the archive is available at `https://groups.google.com/forum/#!forum/vim_dev`.

IRC

In case you're not familiar, **IRC** stands for **Internet Relay Chat**, which is a protocol for exchanging messages. IRC is mainly used for group discussions.

Many Vim core developers and users frequent the Vim IRC channel. The Vim channel is a great place to ask questions and get a general feeling for the Vim community.

You can log in through Freenode's Web client at `https://webchat.freenode.net` or through an IRC client of your choice. Personally, I prefer using the irssi command-line client, but it takes quite a lot of tinkering to get the settings just right.

Other communities

There are a lot more active communities on the web. Here are a couple of highlights:

- An active Reddit community can be found at `https://reddit.com/r/vim`
- A Vim Q&A site is available at `https://vi.stackexchange.com/`

Learning resources

Everyone learns differently, and it's hard to recommend a resource that will work for everyone. Here are a few resources I found helpful:

- Vim Tips Wiki is a huge repository of bite-sized Vim tips: `https://vim.wikia.com`
- Vim screencasts: `http://vimcasts.org`
- *Learn Vimscript the Hard Way* is a fantastic in-depth Vimscript tutorial: `http://learnvimscriptthehardway.stevelosh.com`
- Sites such as `https://vim-adventures.com/` and `https://www.vimgolf.com/` are another fun way to continue your Vim learning journey

Bram Moolenaar, the original creator of Vim, had a personal website with a few Vim-related notes: `https://moolenaar.net`. Bram was actively involved in a non-profit making organization helping kids in Uganda, and you can head over to his home page to learn more about it. Unfortunately, Bram passed away in 2023.

Finally, I sometimes post Vim-related snippets on my blog at `https://www.rosipov.com`. It's usually filled with unrelated articles, but you can filter to only display Vim-related posts: `https://www.rosipov.com/blog/categories/vim`.

A word about Uganda

Vim is free software, but its creator and many core contributors encourage users to donate to the **International Child Care Fund** (**ICCF**) Holland. This charity supports disadvantaged children in Uganda. To learn more about ICCF and Vim's connection to this cause, run `:help Uganda` within Vim.

Summary

In the final chapter of this book, we've looked at *Seven habits of effective text editing* – Bram Moolenaar's article, which primes you to detect inefficiencies in your workflow, correct them, and turn them into a habit.

We've exposed some ways in which you can continue using Vi-like editing experience in other IDEs and text editors, web browsers (through the likes of **Vimium**), and everywhere else (through **vim-anywhere** or **Text Editor Anywhere**).

We've covered some of the ways to get in touch with other Vim users and developers: through mailing lists, IRC channels, Reddit, and other mediums. We've also touched on some learning resources, including **Vim Tips Wiki** and **Learn Vimscript the Hard Way**.

Happy Vimming!

Index

D

E

F

G

H

I

L

M

N

U

V

W

Y

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Mastering Vim*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-83508-187-7`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly