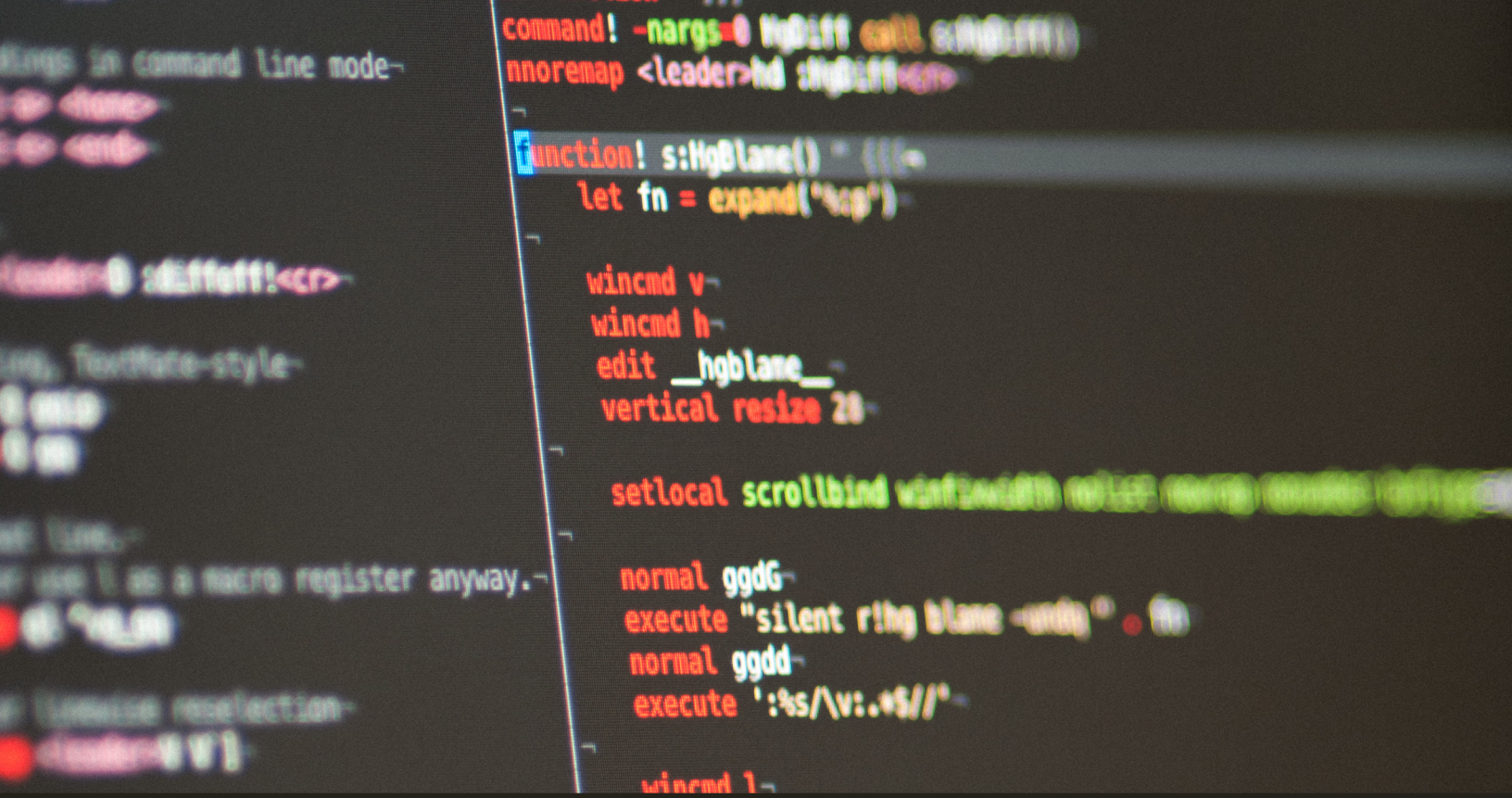
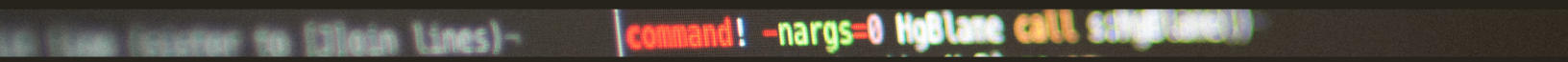
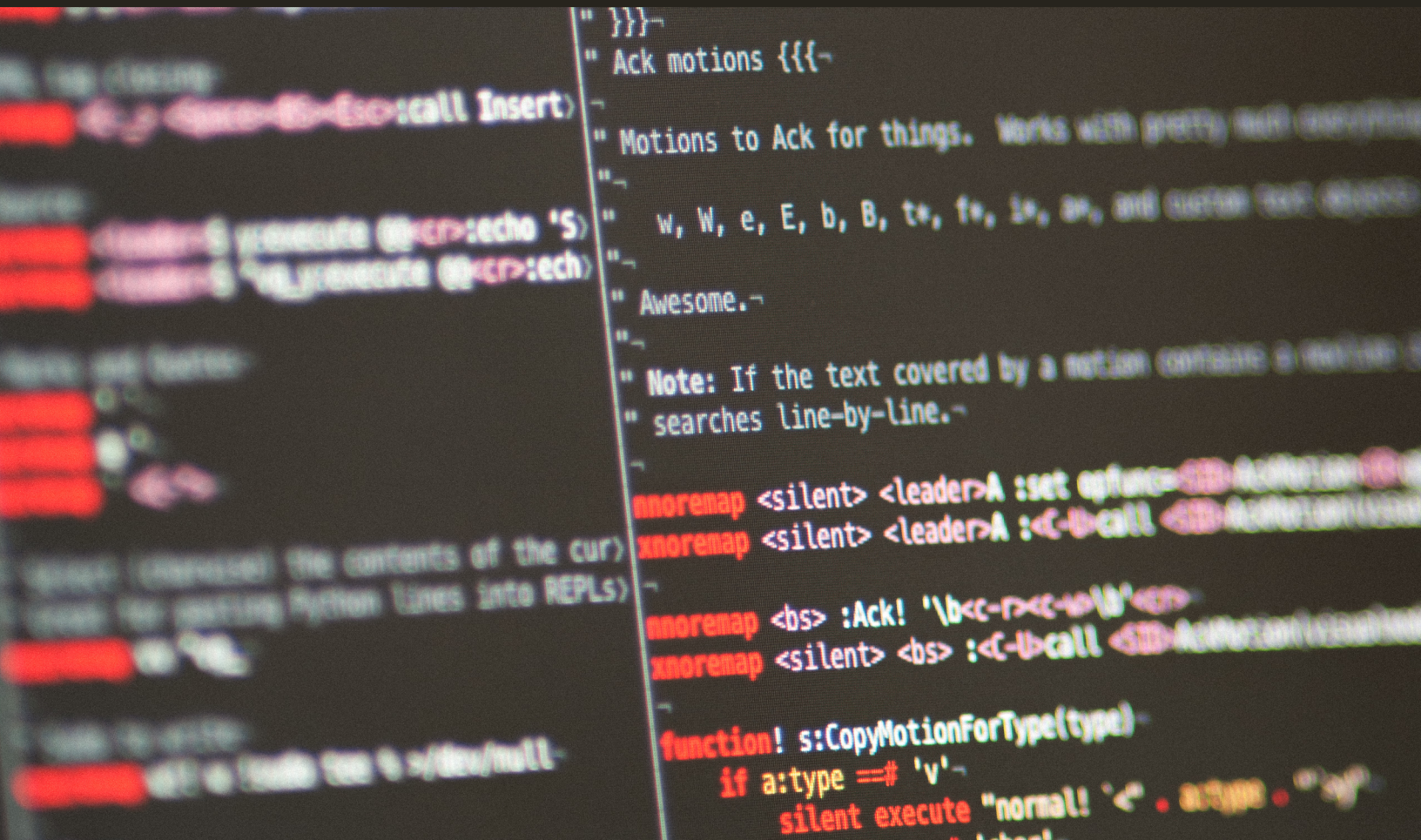


Learn Vimscript the Hard Way



Steve Losh



Learn Vimscript the Hard Way

Steve Losh

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

This version was published on 2013-04-04

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

Contents

Preface

Programmers shape ideas into text.

That text gets turned into numbers and those numbers bump into other numbers and *make things happen*.

As programmers, we use text editors to get our ideas out of our heads and create the chunks of text we call "programs". Full-time programmers will spend tens of thousands of hours of their lives interacting with their text editor, during which they'll be doing many things:

- Getting raw text from their brains into their computers.
- Correcting mistakes in that text.
- Restructuring the text to formulate a problem in a different way.
- Documenting how and why something was done a particular way.
- Communicating with other programmers about all of these things.

Vim is incredibly powerful out of the box, but it doesn't truly shine until you take some time to customize it for your particular work, habits, and fingers. This book will introduce you to Vimscript, the main programming language used to customize Vim. You'll be able to mold Vim into an editor suited to your own personal text editing needs and make the rest of your time in Vim more efficient.

Along the way I'll also mention things that aren't strictly about Vimscript, but are more about learning and being more efficient in general. Vimscript isn't going to help you much if you wind up fiddling with your editor all day instead of working, so it's important to strike a balance.

The style of this book is a bit different from most other books about programming languages. Instead of simply presenting you with facts about how Vimscript works, it guides you through typing in commands to see what they do.

Sometimes the book will lead you into dead ends before explaining the "right way" to solve a problem. Most other books don't do this, or only mention the sticky issues *after* showing you the solution. This isn't how things typically happen in the real world, though. Often you'll be writing a quick piece of Vimscript and run into a quirk of the language that you'll need to figure out. By stepping through this process in the book instead of glossing over it I hope to get you used to dealing with Vimscript's peculiarities so you're ready when you find edge cases of your own. Practice makes perfect.

Each chapter of the book focuses on a single topic. They're short but packed with information, so don't just skim them. If you really want to get the most out of this book you need to actually type in all of the commands. You may already be an experienced programmer who's used to reading code and understanding it straight away. If so: it doesn't matter. Learning Vim and Vimscript is a different experience from learning a normal programming language.

You need to **type in *all* the commands**.

You need to **do *all* the exercises**.

There are two reasons this is so important. First, Vimscript is old and has a lot of dusty corners and twisty hallways. One configuration option can change how the entire language works. By typing *every* command in *every* lesson and doing *every* exercise you'll discover problems with your Vim build or configuration on the simpler commands, where they'll be easier to diagnose and fix.

Second, Vimscript *is* Vim. To save a file in Vim, you type `:write` (or `:w` for short) and press return. To save a file in a Vimscript, you use `write`. Many of the Vimscript commands you'll learn can be used in your day-to-day editing as well, but they're only helpful if they're in your muscle memory, which simply doesn't happen from just reading.

I hope you'll find this book useful. It's *not* meant to be a comprehensive guide to Vimscript. It's meant to get you comfortable enough with the language to mold Vim to your taste, write some simple plugins for other users, read other people's code (with regular side-trips to `:help`), and recognize some of the common pitfalls.

Good luck!

Acknowledgements

First I'd like to thank Zed Shaw[1] for writing Learn Python the Hard Way[2] and making it freely available. This book's format and writing style is directly inspired by it.

I'd also like to thank the following GitHub and Bitbucket users who sent pull requests, pointed out typos, raised issues, and otherwise contributed:

- aperiodic[3]
- billturner[4]
- chiphogg[5]
- ciwchris[6]
- cwarden[7]
- dmedvinsky[8]
- flatcap[9]
- helixbass[10]
- hoelzro[11]
- jrib[12]
- lheiskan[13]
- lightningdb[14]
- manojkumarm[15]
- manojkumarm[16]
- markscholtz[17]
- marlun[18]
- mattsacks[19]

[1] http://zedshaw.com/
[2] http://learnpythonthehardway.org/
[3] https://github.com/aperiodic
[4] https://github.com/billturner
[5] https://github.com/chiphogg
[6] https://github.com/ciwchris
[7] https://github.com/cwarden
[8] https://github.com/dmedvinsky
[9] https://github.com/flatcap
[10] https://bitbucket.org/helixbass
[11] https://github.com/hoelzro
[12] https://github.com/jrib
[13] https://github.com/lheiskan
[14] https://github.com/lightningdb
[15] https://github.com/manojkumarm
[16] https://github.com/manojkumarm
[17] https://github.com/markscholtz
[18] https://github.com/marlun
[19] https://github.com/mattsacks

- Mr-Happy[20]
- mrgrubb[21]
- NagatoPain[22]
- nathanaelkane[23]
- nielsbom[24]
- nvie[25]
- Psycojoker[26]
- riceissa[27]
- rodnaph[28]
- rramsden[29]
- sedm0784[30]
- sherrillmix[31]
- tapichu[32]
- ZyX-I[33]

I apologize to anyone I've forgotten.

[20] https://github.com/Mr-Happy
[21] https://github.com/mrgrubb
[22] https://github.com/NagatoPain
[23] https://github.com/nathanaelkane
[24] https://github.com/nielsbom
[25] https://github.com/nvie
[26] https://github.com/Psycojoker
[27] https://github.com/riceissa
[28] https://github.com/rodnaph
[29] https://github.com/rramsden
[30] https://github.com/sedm0784
[31] https://github.com/sherrillmix
[32] https://github.com/tapichu
[33] https://github.com/ZyX-I

Prerequisites

To use this book you should have the latest version of Vim installed, which is version 7.3 at the time of this writing. New versions of Vim are almost always backwards-compatible, so everything in this book should work fine with anything after 7.3 too.

Nothing in this book is specific to console Vim or GUI Vims like gVim or MacVim. You can use whichever you prefer.

You should be comfortable editing files in Vim. You should know basic Vim terminology like "buffer", "window", "normal mode", "insert mode" and "text object".

If you're not at that point yet you should go through the `vimtutor` program, use Vim exclusively for a month or two, and come back when you've got Vim burned into your fingers.

You'll also need to have some programming experience. If you've never programmed before check out Learn Python the Hard Way[34] first and come back to this book when you're done.

Creating a Vimrc File

If you already know what a `~/.vimrc` file is and have one, go on to the next chapter.

A `~/.vimrc` file is a file you create that contains some Vimscript code. Vim will automatically run the code inside this file every time you open Vim.

On Linux and Mac OS X this file is located in your home directory and named `.vimrc`.

On Windows this file is located in your home folder and named `_vimrc`.

To easily find the location and name of the file on *any* operating system, run `:echo $MYVIMRC` in Vim. The path will be displayed at the bottom of the screen.

Create this file if it doesn't already exist.

[34] http://learnpythonthehardway.org/

1 Echoing Messages

The first pieces of Vimscript we'll look at are the `echo` and `echom` commands.

You can read their full documentation by running `:help echo` and `:help echom` in Vim. As you go through this book you should try to read the `:help` for every new command you encounter to learn more about them.

Try out `echo` by running the following command:

```
:echo "Hello, world!"
```

You should see `Hello, world!` appear at the bottom of the window.

1.1 Persistent Echoing

Now try out `echom` by running the following command.

```
:echom "Hello again, world!"
```

You should see `Hello again, world!` appear at the bottom of the window.

To see the difference between these two commands, run the following:

```
:messages
```

You should see a list of messages. `Hello, world!` will *not* be in this list, but `Hello again, world!` *will* be in it.

When you're writing more complicated Vimscript later in this book you may find yourself wanting to "print some output" to help you debug problems. Plain old `:echo` will print output, but it will often disappear by the time your script is done. Using `:echom` will save the output and let you run `:messages` to view it later.

1.2 Comments

Before moving on, let's look at how to add comments. When you write Vimscript code (in your `~/.vimrc` file or any other one) you can add comments with the `"` character, like this:

```
" Make space more useful
nnoremap <space> za
```

This doesn't *always* work (that's one of those ugly corners of Vimscript), but in most cases it does. Later we'll talk about when it won't (and why that happens).

1.3 Exercises

Read `:help echo`.

Read `:help echom`.

Read `:help messages`.

Add a line to your `~/.vimrc` file that displays a friendly ASCII-art cat (`>^.^<`) whenever you open Vim.

2 Setting Options

Vim has many options you can set to change how it behaves.

There are two main kinds of options: boolean options (either "on" or "off") and options that take a value.

2.1 Boolean Options

Run the following command:

```
:set number
```

Line numbers should appear on the left side of the window if they weren't there already. Now run this:

```
:set nonumber
```

The line numbers should disappear. `number` is a boolean option: it can be off or on. You turn it "on" by running `:set number` and "off" with `:set nonumber`.

All boolean options work this way. `:set <name>` turns the option on and `:set no<name>` turns it off.

2.2 Toggling Boolean Options

You can also "toggle" boolean options to set them to the *opposite* of whatever they are now. Run this:

```
:set number!
```

The line numbers should reappear. Now run it again:

```
:set number!
```

They should disappear once more. Adding a `!` (exclamation point or "bang") to a boolean option toggles it.

2.3 Checking Options

You can ask Vim what an option is currently set to by using a `?`. Run these commands and watch what happens after each:

```
:set number
:set number?
:set nonumber
:set number?
```

Notice how the first `:set number?` command displayed `number` while the second displayed `nonumber`.

2.4 Options with Values

Some options take a value instead of just being off or on. Run the following commands and watch what happens after each:

```
:set number
:set numberwidth=10
:set numberwidth=4
:set numberwidth?
```

The `numberwidth` option changes how wide the column containing line numbers will be. You can change non-boolean options with `:set <name>=<value>`, and check them the usual way (`:set <name>?`).

Try checking what a few other common options are set to:

```
:set wrap?
:set shiftround?
:set matchtime?
```

2.5 Setting Multiple Options at Once

Finally, you can specify more than one option in the same `:set` command to save on some typing. Try running this:

```
:set numberwidth=2
:set nonumber
:set number numberwidth=6
```

Notice how both options were set and took effect in the last command.

2.6 Exercises

Read `:help 'number'` (notice the quotes).

Read `:help relativenumber`.

Read `:help numberwidth`.

Read `:help wrap`.

Read `:help shiftround`.

Read `:help matchtime`.

Add a few lines to your `~/.vimrc` file to set these options however you like.

3 Basic Mapping

If there's one feature of Vimscript that will let you bend Vim to your will more than any other, it's the ability to map keys. Mapping keys lets you tell Vim:

> When I press this key, I want you to do this stuff instead of whatever you would normally do.

We're going to start off by mapping keys in normal mode. We'll talk about how to map keys in insert and other modes in the next chapter.

Type a few lines of text into a file, then run:

```
:map - x
```

Put your cursor somewhere in the text and press `-`. Notice how Vim deleted the character under the cursor, just like if you had pressed `x`.

We already have a key for "delete the character under the cursor", so let's change that mapping to something slightly more useful. Run this command:

```
:map - dd
```

Now put your cursor on a line somewhere and press `-` again. This time Vim deletes the entire line, because that's what `dd` does.

3.1 Special Characters

You can use `<keyname>` to tell Vim about special keys. Try running this command:

```
:map <space> viw
```

Put your cursor on a word in your text and press the space bar. Vim will visually select the word.

You can also map modifier keys like Ctrl and Alt. Run this:

```
:map <c-d> dd
```

Now pressing `Ctrl+d` on your keyboard will run `dd`.

3.2 Commenting

Remember in the first lesson where we talked about comments? Mapping keys is one of the places where Vim comments don't work. Try running this command:

```
:map <space> viw " Select word
```

If you try pressing space now, something horrible will almost certainly happen. Why?

When you press the space bar now, Vim thinks you want it to do what `viw<space>"<space>Select<space>word` would do. Obviously this isn't what we want.

If you look closely at the effect of this mapping you might notice something strange. Take a few minutes to try to figure out exactly what happens when you use it, and *why* that happens.

Don't worry if you don't get it right away – we'll talk about it more soon.

3.3 Exercises

Map the - key to "delete the current line, then paste it below the one we're on now". This will let you move lines downward in your file with one keystroke.

Add that mapping command to your `~/.vimrc` file so you can use it any time you start Vim.

Figure out how to map the _ key to move the line up instead of down.

Add that mapping to your `~/.vimrc` file too.

Try to guess how you might remove a mapping and reset a key to its normal function.

4 Modal Mapping

In the last chapter we talked about how to map keys in Vim. We used the `map` command which made the keys work in normal mode. If you played around a bit before moving on to this chapter, you may have noticed that the mappings also took effect in visual mode.

You can be more specific about when you want mappings to apply by using `nmap`, `vmap`, and `imap`. These tell Vim to only use the mapping in normal, visual, or insert mode respectively.

Run this command:

```
:nmap \ dd
```

Now put your cursor in your text file, make sure you're in normal mode, and press `\`. Vim will delete the current line.

Now enter visual mode and try pressing `\`. Nothing will happen, because we told Vim to only use that mapping in normal mode (and `\` doesn't do anything by default).

Run this command:

```
:vmap \ U
```

Enter visual mode and select some text, then press `\`. Vim will convert the text to uppercase!

Try the `\` key a few times in normal and visual modes and notice that it now does something completely different depending on which mode you're in.

4.1 Muscle Memory

At first the idea of mapping the same key to do different things depending on which mode you're in may sound like a terrible idea. Why would you want to have to stop and think which mode you're in before pressing the key? Wouldn't that negate any time you save from the mapping itself?

In practice it turns out that this isn't really a problem. Once you start using Vim often you won't be thinking about the individual keys you're typing any more. You'll think: "delete a line" and not "press `dd`". Your fingers and brain will learn your mappings and the keys themselves will become subconscious.

4.2 Insert Mode

Now that we've covered how to map keys in normal and visual mode, let's move on to insert mode. Run this command:

```
:imap <c-d> dd
```

You might think that this would let you press `Ctrl+d` whenever you're in insert mode to delete the current line. This would be handy because you wouldn't need to go back into normal mode to cut out lines.

Go ahead and try it. It won't work – instead it will just put two `d`s in your file! That's pretty useless.

The problem is that Vim is doing exactly what we told it to. We said: "when I press `<c-d>` I want you to do what pressing `d` and `d` would normally do". Well, normally when you're in insert mode and press the `d` key twice, you get two `d`s in a row!

To make this mapping do what we intended we need to be very explicit. Run this command to change the mapping:

```
:imap <c-d> <esc>dd
```

The `<esc>` is our way of telling Vim to press the Escape key, which will take us out of insert mode.

Now try the mapping. It works, but notice how you're now back in normal mode. This makes sense because we told Vim that `<c-d>` should exit insert mode and delete a line, but we never told it to go back into insert mode.

Run one more command to fix the mapping once and for all:

```
:imap <c-d> <esc>ddi
```

The `i` at the end enters insert mode, and our mapping is finally complete.

4.3 Exercises

Set up a mapping so that you can press `<c-u>` to convert the current word to uppercase when you're in insert mode. Remember that `U` in visual mode will uppercase the selection. I find this mapping extremely useful when I'm writing out the name of a long constant like `MAX_CONNECTIONS_ALLOWED`. I type out the constant in lower case and then uppercase it with the mapping instead of holding shift the entire time.

Add that mapping to your `~/.vimrc` file.

Set up a mapping so that you can uppercase the current word with `<c-u>` when in *normal* mode. This will be slightly different than the previous mapping because you don't need to enter normal mode. You should end up back in normal mode at the end instead of in insert mode as well.

Add that mapping to your `~/.vimrc` file.

5 Strict Mapping

Get ready, because things are about to get a little wild.

So far we've used `map`, `nmap`, `vmap`, and `imap` to create key mappings that will save time. These work, but they have a downside. Run the following commands:

```
:nmap - dd
:nmap \ -
```

Now try pressing `\` (in normal mode). What happens?

When you press `\` Vim sees the mapping and says "I should run `-` instead". But we've already mapped `-` to do something else! Vim sees that and says "oh, now I need to run `dd`", and so it deletes the current line.

When you map keys with these commands Vim will take *other* mappings into account. This may sound like a good thing at first but in reality it's pure evil. Let's talk about why, but first remove those mappings by running the following commands:

```
:nunmap -
:nunmap \
```

5.1 Recursion

Run this command:

```
:nmap dd O<esc>jddk
```

At first glance it might look like this would map `dd` to:

- Open a new line above this one.
- Exit insert mode.
- Move back down.
- Delete the current line.
- Move up to the blank line just created.

Effectively this should "clear the current line". Try it.

Vim will seem to freeze when you press `dd`. If you press `<c-c>` you'll get Vim back, but there will be a ton of empty lines in your file! What happened?

This mapping is actually *recursive*! When you press `dd`, Vim says:

- `dd` is mapped, so perform the mapping.
    - Open a line.
    - Exit insert mode.
    - Move down a line.
    - `dd` is mapped, so perform the mapping.
        * Open a line.
        * Exit insert mode.
        * Move down a line.
        * `dd` is mapped, so perform the mapping, and so on.

This mapping can never finish running! Go ahead and remove this terrible thing with the following command:

```
:nunmap dd
```

5.2 Side Effects

One downside of the `*map` commands is the danger of recursing. Another is that their behavior can change if you install a plugin that maps keys they depend on.

When you install a new Vim plugin there's a good chance that you won't use and memorize every mapping it creates. Even if you do, you'd have to go back and look through your `~/.vimrc` file to make sure none of your custom mappings use a key that the plugin has mapped.

This would make installing plugins tedious and error-prone. There must be a better way.

5.3 Nonrecursive Mapping

Vim offers another set of mapping commands that will *not* take mappings into account when they perform their actions. Run these commands:

```
:nmap x dd
:nnoremap \ x
```

Now press `\` and see what happens.

When you press `\` Vim ignores the `x` mapping and does whatever it would do for `x` by default. Instead of deleting the current line, it deletes the current character.

Each of the `*map` commands has a `*noremap` counterpart that ignores other mappings: `noremap`, `nnoremap`, `vnoremap`, and `inoremap`.

When should you use these nonrecursive variants instead of their normal counterparts?

Always.

No, seriously, *always*.

Using a bare `*map` is just *asking* for pain down the road when you install a plugin or add a new custom mapping. Save yourself the trouble and type the extra characters to make sure it never happens.

5.4 Exercises

Convert all the mappings you added to your `~/.vimrc` file in the previous chapters to their nonrecursive counterparts.

Read `:help unmap`.

6 Leaders

We've learned how to map keys in a way that won't make us want to tear our hair out later, but you might have noticed one more problem.

Every time we do something like `:nnoremap <space> dd` we've overwritten what `<space>` normally does. What if we need that key later?

There are a bunch of keys that you don't normally need in your day-to-day Vim usage. `-`, `H`, `L`, `<space>`, `<cr>`, and `<bs>` do things that you almost never need (in normal mode, of course). Depending on how you work you may find others that you never use.

Those are safe to map, but that only gives us six keys to work with. What happened to Vim's legendary customizability?

6.1 Mapping Key Sequences

Unlike Emacs, Vim makes it easy to map more than just single keys. Run these commands:

```
:nnoremap -d dd
:nnoremap -c ddO
```

Try them out by typing `-d` and `-c` (quickly) in normal mode. The first creates a custom mapping to delete a line, while the second "clears" a line and puts you into insert mode.

This means you can pick a key that you don't care about (like `-`) as a "prefix" key and create mappings on top of it. It means you'll have to type an extra key to activate the mappings, but one extra keystroke can easily be absorbed into muscle memory.

If you think this might be a good idea, you're right, and it turns out that Vim already has mechanisms for this "prefix" key!

6.2 Leader

Vim calls this "prefix" key the "leader". You can set your leader key to whatever you like. Run this command:

```
:let mapleader = "-"
```

You can replace `-` with any key you like. I personally like `,` even though it shadows a useful function, because it's very easy to type.

When you're creating new mappings you can use `<leader>` to mean "whatever I have my leader key set to". Run this command:

```
:nnoremap <leader>d dd
```

Now try it out by pressing your leader key and then `d`. Vim will delete the current line.

Why bother with setting `<leader>` at all, though? Why not just include your "prefix" key directly in your mapping commands? There are three good reasons.

First of all, you may decide you need the normal function of your leader later on down the road. Defining it in one place makes it easy to change later.

Second, when someone else is looking at your `~/.vimrc` file they'll immediately know what you mean when you say `<leader>`. They can simply copy your mapping into their own `~/.vimrc` if they like it even if they use a different leader.

Finally, many Vim plugins create mappings that start with `<leader>`. If you've already got it set up they'll work properly and will feel familiar right out of the box.

6.3 Local Leader

Vim has a second "leader" key called "local leader". This is meant to be a prefix for mappings that only take effect for certain types of files, like Python files or HTML files.

We'll talk about how to make mappings for specific types of files later in the book, but you can go ahead and set your "localleader" now:

```
:let maplocalleader = "\\"
```

Notice that we have to use `\\` and not just `\` because `\` is the escape character in Vimscript strings. You'll learn more about this later.

Now you can use `<localleader>` in mappings and it will work just like `<leader>` does (except for resolving to a different key, of course).

Feel free to change this key to something else if you don't like backslash.

6.4 Exercises

Read `:help mapleader`.

Read `:help maplocalleader`.

Set `mapleader` and `maplocalleader` in your `~/.vimrc` file.

Convert all the mappings you added to your `~/.vimrc` file in the previous chapters to be prefixed with `<leader>` so they don't shadow existing commands.

7 Editing Your Vimrc

Before we move on to learning more Vimscript, let's find a way to make it easier to add new mappings to our ~/.vimrc file.

Sometimes you're coding away furiously at a problem and realize a new mapping would make your editing easier. You should add it to your ~/.vimrc file right then and there to make sure you don't forget, but you *don't* want to lose your concentration.

The idea of this chapter is that you want to make it easier to make it easier to edit text.

That's not a typo. Read it again.

The idea of this chapter is that you want to (make it easier to (make it easier to (edit text))).

7.1 Editing Mapping

Let's add a mapping that will open your ~/.vimrc file in a split so you can quickly edit it and get back to coding. Run this command:

```
:nnoremap <leader>ev :vsplit $MYVIMRC<cr>
```

I like to think of this command as "**e**dit my **v**imrc file".

$MYVIMRC is a special Vim variable that points to your ~/.vimrc file. Don't worry about that for right now, just trust me that it works.

:vsplit opens a new vertical split. If you'd prefer a horizontal split you can replace it with :split.

Take a minute and think through that command in your mind. The goal is: "open my ~/.vimrc file in a new split". Why does it work? Why is every single piece of that mapping necessary?

With that mapping you can open up your ~/.vimrc file with three keystrokes. Once you use it a few times it will burn its way into your muscle memory and take less than half a second to type.

When you're in the middle of coding and come up with a new mapping that would save you time it's now trivial to add it to your ~/.vimrc file.

7.2 Sourcing Mapping

Once you've added a mapping to your ~/.vimrc file, it doesn't immediately take effect. Your ~/.vimrc file is only read when you start Vim. This means you need to also run the command manually to make it work in the current session, which is a pain.

Let's add a mapping to make this easier:

```
:nnoremap <leader>sv :source $MYVIMRC<cr>
```

I like to think of this command as "source my vimrc file".

The `source` command tells Vim to take the contents of the given file and execute it as Vimscript.

Now you can easily add new mappings during the heat of coding:

- Use `<leader>ev` to open the file.
- Add the mapping.
- Use `:wq<cr>` (or `ZZ`) to write the file and close the split, bringing you back to where you were.
- Use `<leader>sv` to source the file and make our changes take effect.

That's eight keystrokes plus whatever it takes to define the mapping. It's very little overhead, which reduces the chance of breaking focus.

7.3 Exercises

Add mappings to "edit my `~/.vimrc`" and "source my `~/.vimrc`" to your `~/.vimrc` file.

Try them out a few times, adding dummy mappings each time.

Read `:help myvimrc`.

8 Abbreviations

Vim has a feature called "abbreviations" that feel similar to mappings but are meant for use in insert, replace, and command modes. They're extremely flexible and powerful, but we're just going to cover the most common uses here.

We're only going to worry about insert mode abbreviations in this book. Run the following command:

```
:iabbrev adn and
```

Now enter insert mode and type:

```
One adn two.
```

As soon as you hit space after typing the `adn` Vim will replace it with `and`.

Correcting typos like this is a great use for abbreviations. Run these commands:

```
:iabbrev waht what
:iabbrev tehn then
```

Now enter insert mode again and type:

```
Well, I don't know waht we should do tehn.
```

Notice how *both* abbreviations were substituted, even though you didn't type a space after the second one.

8.1 Keyword Characters

Vim will substitute an abbreviation when you type any "non-keyword character" after an abbreviation. "Non-keyword character" means any character not in the `iskeyword` option. Run this command:

```
:set iskeyword?
```

You should see something like `iskeyword=@,48-57,_,192-255`. This format is very complicated, but in essence it means that all of the following are considered "keyword characters":

- The underscore character (_).
- All alphabetic ASCII characters, both upper and lower case, and their accented versions.
- Any characters with an ASCII value between 48 and 57 (the digits zero through nine).
- Any characters with an ASCII value between 192 and 255 (some special ASCII characters).

If you want to read the *full* description of this option's format you can check out `:help isfname`, but I'll warn you that you'd better have a beer at the ready for this one.

For our purposes you can simply remember that abbreviations will be expanded when you type anything that's not a letter, number, or underscore.

8.2 More Abbreviations

Abbreviations are useful for more than just correcting typos. Let's add a few more that can help in day-to-day text editing. Run the following commands:

```
:iabbrev @@    steve@stevelosh.com
:iabbrev ccopy Copyright 2013 Steve Losh, all rights reserved.
```

Feel free to replace my name and email address with your own, then enter insert mode and try them out.

These abbreviations take large chunks of text that you type often and compress them down to a few characters. Over time, this can save you a lot of typing, as well as wear and tear on your fingers.

8.3 Why Not Use Mappings?

If you're thinking that abbreviations seem similar to mappings, you're right. However, they're intended to be used for different things. Let's look at an example.

Run this command:

```
:inoremap ssig -- <cr>Steve Losh<cr>steve@stevelosh.com
```

This is a *mapping* intended to let you insert your signature quickly. Try it out by entering insert mode and typing `ssig`.

It seems to work great, but there's a problem. Try entering insert mode and typing this text:

```
Larry Lessig wrote the book "Remix".
```

You'll notice that Vim has expanded the `ssig` in Larry's name! Mappings don't take into account what characters come before or after the map – they only look at the specific sequence that you mapped to.

Remove the mapping and replace it with an abbreviation by running the following commands:

```
:iunmap ssig
:iabbrev ssig -- <cr>Steve Losh<cr>steve@stevelosh.com
```

Now try out the abbreviation again.

This time Vim will pay attention to the characters before and after `ssig` and only expand it when we want.

8.4 Exercises

Add abbreviations for some common typos you know you personally make to your `~/.vimrc` file. Be sure to use the mappings you created in the last chapter to open and source the file!

Add abbreviations for your own email address, website, and signature as well.

Think of some pieces of text you type very often and add abbreviations for them too.

9 More Mappings

I know we've talked a lot about mappings so far, but we're going to practice them again now. Mappings are one of the easiest and fastest ways to make your Vim editing more productive so it's good to focus on them quite a bit.

One concept that has showed up in several examples but that we haven't explicitly talked about is mapping a sequence of multiple keys.

Run the following command:

```
:nnoremap jk dd
```

Now make sure you're in normal mode and press `j` followed quickly by `k`. Vim will delete the current line.

Now try pressing only `j` and waiting for a bit. If you don't press `k` quickly after the `j`, Vim decides that you don't want to activate the mapping and instead runs the normal `j` functionality (moving down a line).

This mapping will make it painful to move around, so let's remove it. Run the following command:

```
:nunmap jk
```

Now typing `jk` in normal mode will move down and then up a line as usual.

9.1 A More Complicated Mapping

You've seen a bunch of simple mappings so far, so it's time to look at something with a bit more meat to it. Run the following command:

```
:nnoremap <leader>" viw<esc>a"<esc>hbi"<esc>lel
```

Now *that's* an interesting mapping! First, go ahead and try it out. Enter normal mode, put your cursor over a word in your text and type `<leader>"`. Vim will surround the word in double quotes!

How does this work? Let's split it apart into pieces and think of what each one does:

```
viw<esc>a"<esc>hbi"<esc>lel
```

- `viw`: visually select the current word
- `<esc>`: exit visual mode, which leaves the cursor on the last character of the word

- `a`: enter insert mode *after* the current character
- `"`: insert a `"` into the text, because we're in insert mode
- `<esc>`: return to normal mode
- `h`: move left one character
- `b`: move back to the beginning of the word
- `i`: enter insert mode *before* the current character
- `"`: insert a `"` into the text again
- `<esc>`: return to normal mode
- `l`: move right, which puts our cursor on the first character of the word
- `e`: move to the end of the word
- `l`: move right, which puts our cursor over the ending quote

Remember: because we used `nnoremap` instead of `nmap` it doesn't matter if you've mapped any of the keys in this sequence to something else. Vim will use the default functionality for all of them.

Hopefully you can see how much potential Vim's mappings have, as well as how unreadable they can become.

9.2 Exercises

Create a mapping similar to the one we just looked at, but for single quotes instead of double quotes.

Try using `vnoremap` to add a mapping that will wrap whatever text you have *visually selected* in quotes. You'll probably need the `` `< `` and `` `> `` commands for this, so read up on them with `:help `<`.

Map `H` in normal mode to go to the beginning of the current line. Since `h` moves left you can think of `H` as a "stronger" `h`.

Map `L` in normal mode to go to the end of the current line. Since `l` moves right you can think of `L` as a "stronger" `l`.

Find out what commands you just overwrote by reading `:help H` and `:help L`. Decide whether you care about them.

Add all of these mappings to your `~/.vimrc` file, making sure to use your "edit my `~/.vimrc`" and "source my `~/.vimrc`" mappings to do so.

10 Training Your Fingers

In this chapter we're going to talk about how to learn Vim more effectively, but we need to do a bit of preparation first.

Let's set up one more mapping that will save more wear on your left hand than any other mapping you ever create. Run the following command:

```
:inoremap jk <esc>
```

Now enter insert mode and type `jk`. Vim will act as if you pressed the escape key and return you to normal mode.

There are a number of ways to exit insert mode in Vim by default:

- `<esc>`
- `<c-c>`
- `<c-[>`

Each of those requires you to stretch your fingers uncomfortably. Using `jk` is great because the keys are right under two of your strongest fingers and you don't have to perform a chord.

Some people prefer using `jj` instead of `jk`, but I prefer `jk` for two reasons:

- It's typed with two separate keys, so you can "roll" your fingers instead of using the same one twice.
- Pressing `jk` in normal mode out of habit will move down and then up, leaving you exactly where you started. Using `jj` in normal mode will move you to a different place in your file.

If you write in a language where `jk` is a frequently used combination of letters (like Dutch) you'll probably want to pick a different mapping.

10.1 Learning the Map

Now that you've got a great new mapping, how can you learn to use it? Chances are you've already got the escape key in your muscle memory, so when you're editing you'll hit it without even thinking.

The trick to relearning a mapping is to *force* yourself to use it by *disabling* the old key(s). Run the following command:

```
:inoremap <esc> <nop>
```

This effectively disables the escape key in insert mode by telling Vim to perform `<nop>` (no operation) instead. Now you *have* to use your `jk` mapping to exit insert mode.

At first you'll forget, type escape and start trying to do something in normal mode and you'll wind up with stray characters in your text. It will be frustrating, but if you stick with it you'll be surprised at how fast your mind and fingers absorb the new mapping. Within an hour or two you won't be accidentally hitting escape any more.

This idea applies to any new mapping you create to replace an old one, and even to life in general. When you want to change a habit, make it harder or impossible to do!

If you want to start cooking meals instead of microwaving TV dinners, don't buy any TV dinners when you go shopping. You'll cook some real food when you get hungry enough.

If you want to quit smoking, always leave your cigarettes in your car's trunk. When you get the urge to have a casual cigarette you'll think of what a pain in the ass it will be to walk out to the car and are less likely to bother doing it.

10.2 Exercises

If you still find yourself using the arrow keys to navigate around Vim in normal mode, map them to `<nop>` to make yourself stop.

If you still use the arrow keys in insert mode, map them to `<nop>` there too. The right way to use Vim is to get out of insert mode as soon as you can and use normal mode to move around.

11 Buffer-Local Options and Mappings

Now we're going to take a few minutes to revisit three things we've already talked about: mappings, abbreviations, and options, but with a twist. We're going to set each of them in a single buffer at a time.

The true power of this idea will become apparent in the next chapter, but we need to lay the groundwork for it now.

For this chapter you'll need to open two files in Vim, each in its own split. I'll call them `foo` and `bar`, but you can name them whatever you like. Put some text into each of them.

11.1 Mappings

Switch to file `foo` and run the following commands:

```
:nnoremap          <leader>d dd
:nnoremap <buffer> <leader>x dd
```

Now stay in file `foo`, make sure you're in normal mode, and type `<leader>d`. Vim will delete a line. This is nothing new.

Still in file `foo`, type `<leader>x`. Vim will delete a line again. This makes sense because we mapped `<leader>x` to `dd` as well.

Now move over to file `bar`. While in normal mode, type `<leader>d`. Again, Vim deletes the current line. Nothing surprising here either.

Now for the twist: while still in file `bar`, type `<leader>x`.

Instead of deleting the entire line, Vim just deleted a single character! What happened?

The `<buffer>` in the second `nnoremap` command told Vim to only consider that mapping when we're in the buffer where we defined it.

When you typed `<leader>x` in file `bar` Vim couldn't find a mapping that matched it, so it treated it as two commands: `<leader>` (which does nothing on its own) and `x` (the normal command to delete a single character.)

11.2 Local Leader

In our example we used `<leader>x` for our buffer-local mapping, but this is bad form. In general, when you create a mapping that only applies to specific buffers you should use `<localleader>` instead of `<leader>`.

Using two separate leader keys provides a sort of "namespacing" that will help you keep all your various mappings straight in your head.

It's even more important when you're writing a plugin for other people to use. The convention of using `<localleader>` for local mappings will prevent your plugin from overwriting someone else's `<leader>` mapping that they've painstakingly burned into their fingers over time.

11.3 Settings

In one of the earliest chapters of the book we talked about settings options with `set`. Some options always apply to all of Vim, but others can be set on a per-buffer basis.

Switch to file `foo` and run the following command:

```
:setlocal wrap
```

Now switch to file `bar` and run this command:

```
:setlocal nowrap
```

Make your Vim window smaller and you'll see that the lines in `foo` wrap, but the lines in `bar` don't.

Let's try another option. Switch to `foo` and run this command:

```
:setlocal number
```

Now switch over to `bar` and run this command:

```
:setlocal nonumber
```

You now have line numbers in `foo` but not in `bar`.

Not all options can be used with `setlocal`. To see if you can set a particular option locally, read its `:help`.

I've glossed over a bit of detail about how local options *actually* work for now. In the exercises you'll learn more about the gory details.

11.4 Shadowing

Before we move on, let's look at a particularly interesting property of local mappings. Switch over to `foo` and run the following commands:

```
:nnoremap <buffer> Q x
:nnoremap          Q dd
```

Now type `Q`. What happens?

When you press `Q`, Vim will run the first mapping, not the second, because the first mapping is *more specific* than the second.

Switch to file `bar` and type `Q` to see that Vim uses the second mapping, because it's not shadowed by the first in this buffer.

11.5 Exercises

Read `:help local-options`.

Read `:help setlocal`.

Read `:help map-local`.

12 Autocommands

Now we're going to look at a topic almost as important as mappings: autocommands.

Autocommands are a way to tell Vim to run certain commands whenever certain events happen. Let's dive right into an example.

Open a new file with `:edit foo` and close it right away with `:quit`. Look on your hard drive and you'll notice that the file is not there. This is because Vim doesn't actually *create* the file until you save it for the first time.

Let's change it so that Vim creates files as soon as you edit them. Run the following command:

```
:autocmd BufNewFile * :write
```

This is a lot to take in, but try it out and see that it works. Run `:edit foo` again, close it with `:quit`, and look at your hard drive. This time the file will be there (and empty, of course).

You'll have to close Vim to remove the autocommand. We'll talk about how to avoid this in a later chapter.

12.1 Autocommand Structure

Let's take a closer look at the autocommand we just created:

```
:autocmd BufNewFile * :write
         ^          ^ ^
         |          | |
         |          | The command to run.
         |          |
         |          A "pattern" to filter the event.
         |
         The "event" to watch for.
```

The first piece of the command is the type of event we want to watch for. Vim offers *many* events to watch. Some of them include:

- Starting to edit a file that doesn't already exist.
- Reading a file, whether it exists or not.
- Switching a buffer's `filetype` setting.
- Not pressing a key on your keyboard for a certain amount of time.

- Entering insert mode.
- Exiting insert mode.

This is just a tiny sample of the available events. There are many more you can use to do lots of interesting things.

The next part of the command is a "pattern" that lets you be more specific about when you want the command to fire. Start up a new Vim instance and run the following command:

```
:autocmd BufNewFile *.txt :write
```

This is almost the same as the last command, but this time it will only apply to files whose names end in `.txt`.

Try it out by running `:edit bar`, then `:quit`, then `:edit bar.txt`, then `:quit`. You'll see that Vim writes the `bar.txt` automatically, but *doesn't* write `bar` because it doesn't match the pattern.

The final part of the command is the command we want to run when the event fires. This is pretty self-explanatory, except for one catch: you can't use special characters like `<cr>` in the command. We'll talk about how to get around this limitation later in the book, but for now you'll just have to live with it.

12.2 Another Example

Let's define another autocommand, this time using a different event. Run the following command:

```
:autocmd BufWritePre *.html :normal gg=G
```

We're getting a bit ahead of ourselves here because we're going to talk about `normal` later in the book, but for now you'll need to bear with me because it's tough to come up with useful examples at this point.

Create a new file called `foo.html`. Edit it with Vim and enter the following text *exactly*, including the whitespace:

```
<html>
<body>
 <p>Hello!</p>
                </body>
                 </html>
```

Now save this file with `:w`. What happened? Vim seems to have reindented the file for us before saving it!

For now I want you to trust me that running `:normal gg=G` will tell Vim to reindent the current file. Don't worry about how that works just yet.

What we *do* want to pay attention to is the autocommand. The event type is `BufWritePre`, which means the event will be checked just before you write *any* file.

We used a pattern of `*.html` to ensure that this command will only fire when we're working on files that end in `.html`. This lets us target our autocommands at specific files, which is a very powerful idea that we'll continue to explore later on.

12.3 Multiple Events

You can create a single autocommand bound to *multiple* events by separating the events with a comma. Run this command:

```
:autocmd BufWritePre,BufRead *.html :normal gg=G
```

This is almost like our last command, except it will also reindent the code whenever we *read* an HTML file as well as when we write it. This could be useful if you have coworkers that don't indent their HTML nicely.

A common idiom in Vim scripting is to pair the `BufRead` and `BufNewFile` events together to run a command whenever you open a certain kind of file, regardless of whether it happens to exist already or not. Run the following command:

```
:autocmd BufNewFile,BufRead *.html setlocal nowrap
```

This will turn line wrapping off whenever you're working on an HTML file.

12.4 FileType Events

One of the most useful events is the `FileType` event. This event is fired whenever Vim sets a buffer's `filetype`.

Let's set up a few useful mappings for a variety of file types. Run the following commands:

```
:autocmd FileType javascript nnoremap <buffer> <localleader>c I//<esc>
:autocmd FileType python     nnoremap <buffer> <localleader>c I#<esc>
```

Open a Javascript file (a file that ends in `.js`), pick a line and type `<localleader>c`. This will comment out the line.

Now open a Python file (a file that ends in `.py`), pick a line and type `<localleader>c`. This will comment out the line, but it will use Python's comment character!

Using autocommands alongside the buffer-local mappings we learned about in the last chapter we can create mappings that are specific to the type of file that we're editing.

This reduces the load on our minds when we're coding. Instead of having to think about moving to the beginning of the line and adding a comment character we can simply think "comment this line".

12.5 Exercises

Skim `:help autocmd-events` to see a list of all the events you can bind autocommands to. You don't need to memorize each one right now. Just try to get a feel for the kinds of things you can do.

Create a few `FileType` autocommands that use `setlocal` to set options for your favorite filetypes just the way you like them. Some options you might like to change on a per-filetype basis are `wrap`, `list`, `spell`, and `number`.

Create a few more "comment this line" autocommands for filetypes you work with often.

Add all of these autocommands to your `~/.vimrc` file. Use your shortcut mappings for editing and sourcing it quickly, of course!

13 Buffer-Local Abbreviations

That last chapter was a monster, so let's tackle something easier. We've seen how to define buffer-local mappings and options, so let's apply the same idea to abbreviations.

Open your `foo` and `bar` files again, switch to `foo`, and run the following command:

```
:iabbrev <buffer> --- &mdash;
```

While still in `foo` enter insert mode and type the following text:

```
Hello --- world.
```

Vim will replace the `---` for you. Now switch to `bar` and try it. It should be no surprise that it's not replaced, because we defined the abbreviation to be local to the `foo` buffer.

13.1 Autocommands and Abbreviations

Let's pair up these buffer-local abbreviations with autocommands to set them to make ourselves a little "snippet" system.

Run the following commands:

```
:autocmd FileType python     :iabbrev <buffer> iff if:<left>
:autocmd FileType javascript :iabbrev <buffer> iff if ()<left>
```

Open a Javascript file and try out the `iff` abbreviation. Then open a Python file and try it there too. Vim will perform the appropriate abbreviation depending on the type of the current file.

13.2 Exercises

Create a few more "snippet" abbreviations for some of the things you type often in specific kinds of files. Some good candidates are `return` for most languages, `function` for javascript, and things like `&ldquo;` and `&rdquo;` for HTML files.

Add these snippets to your `~/.vimrc` file.

Remember: the best way to learn to use these new snippets is to *disable* the old way of doing things. Running `:iabbrev <buffer> return NOPENOPENOPE` will *force* you to use your abbreviation instead. Add these "training" snippets to match all the ones you created to save time.

14 Autocommand Groups

A few chapters ago we learned about autocommands. Run the following command:

```
:autocmd BufWrite * :echom "Writing buffer!"
```

Now write the current buffer with `:write` and run `:messages` to view the message log. You should see the `Writing buffer!` message in the list.

Now write the current buffer again and run `:messages` to view the message log. You should see the `Writing buffer!` message in the list twice.

Now run the exact same autocommand again:

```
:autocmd BufWrite * :echom "Writing buffer!"
```

Write the current buffer one more time and run `:messages`. You will see the `Writing buffer!` message in the list *four* times. What happened?

When you create an autocommand like this Vim has no way of knowing if you want it to replace an existing one. In our case, Vim created two *separate* autocommands that each happen to do the same thing.

14.1 The Problem

Now that you know it's possible to create duplicate autocommands, you may be thinking: "So what? Just don't do that!"

The problem is that sourcing your `~/.vimrc` file rereads the entire file, including any autocommands you've defined! This means that every time you source your `~/.vimrc` you'll be duplicating autocommands, which will make Vim run slower because it executes the same commands over and over.

To simulate this, try running the following command:

```
:autocmd BufWrite * :sleep 200m
```

Now write the current buffer. You may or may not notice a slight sluggishness in Vim's writing time. Now run the command three more times:

```
:autocmd BufWrite * :sleep 200m
:autocmd BufWrite * :sleep 200m
:autocmd BufWrite * :sleep 200m
```

Write the file again. This time the slowness will be more apparent.

Obviously you won't have any autocommands that do nothing but sleep, but the `~/.vimrc` of a seasoned Vim user can easily reach 1,000 lines, many of which will be autocommands. Combine that with autocommands defined in any installed plugins and it can definitely affect performance.

14.2 Grouping Autocommands

Vim has a solution to the problem. The first step is to group related autocommands into named groups.

Open a fresh instance of Vim to clear out the autocommands from before, then run the following commands:

```
:augroup testgroup
:    autocmd BufWrite * :echom "Foo"
:    autocmd BufWrite * :echom "Bar"
:augroup END
```

The indentation in the middle two lines is insignificant. You don't have to type it if you don't want to.

Write a buffer and check `:messages`. You should see both `Foo` and `Bar`. Now run the following commands:

```
:augroup testgroup
:    autocmd BufWrite * :echom "Baz"
:augroup END
```

Try to guess what will happen when you write the buffer again. Once you have a guess in mind, write the buffer and check `:messages` to see if you were correct.

14.3 Clearing Groups

What happened when you wrote the file? Was it what you expected?

If you thought Vim would replace the group, you can see that you guessed wrong. Don't worry, most people think the same thing at first (I know I did).

When you use `augroup` multiple times Vim will *combine* the groups each time.

If you want to *clear* a group you can use `autocmd!` inside the group. Run the following commands:

```
:augroup testgroup
:    autocmd!
:    autocmd BufWrite * :echom "Cats"
:augroup END
```

Now try writing your file and checking `:messages`. This time Vim only echoed `Cats` when you wrote the file.

14.4 Using Autocommands in Your Vimrc

Now that we know how to group autocommands and clear those groups, we can use this to add autocommands to `~/.vimrc` that don't add a duplicate every time we source it.

Add the following to your `~/.vimrc` file:

```
augroup filetype_html
    autocmd!
    autocmd FileType html nnoremap <buffer> <localleader>f Vatzf
augroup END
```

We enter the `filetype_html` group, immediately clear it, define an autocommand, and leave the group. If we source `~/.vimrc` again the clearing will prevent Vim from adding duplicate autocommands.

14.5 Exercises

Go through your `~/.vimrc` file and wrap *every* autocommand you have in groups like this. You can put multiple autocommands in the same group if it makes sense to you.

Try to figure out what the mapping in the last example does.

Read `:help autocmd-groups`.

15 Operator-Pending Mappings

In this chapter we're going to explore one more rabbit hole in Vim's mapping system: "operator-pending mappings". Let's step back for a second and make sure we're clear on vocabulary.

An operator is a command that waits for you to enter a movement command, and then does something on the text between where you currently are and where the movement would take you.

Some examples of operators are `d`, `y`, and `c`. For example:

```
Keys   Operator   Movement
----   --------   -------------
dw     Delete     to next word
ci(    Change     inside parens
yt,    Yank       until comma
```

15.1 Movement Mappings

Vim lets you create new movements that work with all existing commands. Run the following command:

```
:onoremap p i(
```

Now type the following text into a buffer:

```
return person.get_pets(type="cat", fluffy_only=True)
```

Put your cursor on the word "cat" and type `dp`. What happened? Vim deleted all the text inside the parentheses. You can think of this new movement as "parameters".

The `onoremap` command tells Vim that when it's waiting for a movement to give to an operator and it sees `p`, it should treat it like `i(`. When we ran `dp` it was like saying "delete parameters", which Vim translates to "delete inside parentheses".

We can use this new mapping immediately with all operators. Type the same text as before into the buffer (or simply undo the change):

```
return person.get_pets(type="cat", fluffy_only=True)
```

Put your cursor on the word "cat" and type `cp`. What happened? Vim deleted all the text inside the parentheses, but this time it left you in insert mode because you used "change" instead of "delete".

Let's try another example. Run the following command:

```
:onoremap b /return<cr>
```

Now type the following text into a buffer:

```
def count(i):
    i += 1
    print i

    return foo
```

Put your cursor on the `i` in the second line and press `db`. What happened? Vim deleted the entire body of the function, all the way up until the `return`, which our mapping used Vim's normal search to find.

When you're trying to think about how to define a new operator-pending movement, you can think of it like this:

1. Start at the cursor position.
2. Enter visual mode (charwise).
3. … mapping keys go here …
4. All the text you want to include in the movement should now be selected.

It's your job to fill in step three with the appropriate keys.

15.2 Changing the Start

You may have already seen a problem in what we've learned so far. If our movements always have to start at the current cursor position it limits what we can do.

Vim isn't in the habit of limiting what you can do, so of course there's a way around this problem. Run the following command:

```
:onoremap in( :<c-u>normal! f(vi(<cr>
```

This might look frightening, but let's try it out. Enter the following text into the buffer:

```
print foo(bar)
```

Put your cursor somewhere in the word `print` and type `cin(`. Vim will delete the contents of the parentheses and place you in insert mode between them.

You can think of this mapping as meaning "inside next parentheses", and it will perform the operator on the text inside the next set of parentheses on the current line.

Let's make a companion "inside last parentheses" ("previous" would be a better word, but it would shadow the "paragraph" movement). Run the following command:

```
:onoremap il( :<c-u>normal! F)vi(<cr>
```

Try it out on some text of your own to make sure it works.

So how do these mappings work? First, the `<c-u>` is something special that you can ignore for now – just trust me that it needs to be there to make the mappings work in all cases. If we remove that we're left with:

```
:normal! F)vi(<cr>
```

`:normal!` is something we'll talk about in a later chapter, but for now it's enough to know that it is a command used to simulate pressing keys in normal mode. For example, running `:normal! dddd` will delete two lines, just like pressing `dddd`. The `<cr>` at the end of the mapping is what executes the `:normal!` command.

So now we know that the mapping is essentially just running the last block of keys:

```
F)vi(
```

This is fairly simple:

- `F)`: Move backwards to the nearest `)` character.
- `vi(`: Visually select inside the parentheses.

We end up with the text we want to operate on visually selected, and Vim performs the operation on it as normal.

15.3 General Rules

A good way to keep the multiple ways of creating operator-pending mappings straight is the remember the following two rules:

- If your operator-pending mapping ends with some text visually selected, Vim will operate on that text.
- Otherwise, Vim will operate on the text between the original cursor position and the new position.

15.4 Exercises

Create operator-pending mappings for "around next parentheses" and "around last parentheses".

Create similar mappings for in/around next/last for curly brackets.

Read `:help omap-info` and see if you can puzzle out what the `<c-u>` in the examples is for.

16 More Operator-Pending Mappings

The idea of operators and movements is one of the most important concepts in Vim, and it's one of the biggest reasons Vim is so efficient. We're going to practice defining new motions a bit more, because extending this powerful idea makes Vim even *more* powerful.

Let's say you're writing some text in Markdown. If you haven't used Markdown before, don't worry, for our purposes here it's very simple. Type the following into a file:

```
Topic One
=========

This is some text about topic one.

It has multiple paragraphs.

Topic Two
=========

This is some text about topic two.  It has only one paragraph.
```

The lines "underlined" with = characters are treated as headings by Markdown. Let's create some mappings that let us target headings with movements. Run the following command:

```
:onoremap ih :<c-u>execute "normal! ?^==\\+$\r:nohlsearch\rkvg_"<cr>
```

This mapping is pretty complicated, so put your cursor in one of the paragraphs (not the headings) and type `cih`. Vim will delete the heading of whatever section you're in and put you in insert mode ("change inside heading").

It uses some things we've never seen before, so let's look at each piece individually. The first part of the mapping, `:onoremap ih` is just the mapping command that we've seen before, so we'll skip over that. We'll keep ignoring the `<c-u>` for the moment as well.

Now we're looking at the remainder of the line:

```
:execute "normal! ?^==\\+$\r:nohlsearch\rkvg_"<cr>
```

16.1 Normal

The `:normal` command takes a set of characters and performs whatever action they would do if they were typed in normal mode. We'll go into greater detail in a later chapter, but we've seen it a few times already so it's time to at least get a taste. Run this command:

```
:normal gg
```

Vim will move you to the top of the file. Now run this command:

```
:normal >>
```

Vim will indent the current line.

For now, don't worry about the `!` after `normal` in our mapping. We'll talk about that later.

16.2 Execute

The `execute` command takes a Vimscript string (which we'll cover in more detail later) and performs it as a command. Run this:

```
:execute "write"
```

Vim will write your file, just as if you had typed `:write<cr>`. Now run this command:

```
:execute "normal! gg"
```

Vim will run `:normal! gg`, which as we just saw will move you to the top of the file. But why bother with this when we could just run the `normal!` command itself?

Look at the following command and try to guess what it will do:

```
:normal! gg/a<cr>
```

It seems like it should:

- Move to the top of the file.
- Start a search.
- Fill in "a" as the target to search for.
- Press return to perform the search.

Run it. Vim will move to the top of the file and nothing else!

The problem is that `normal!` doesn't recognize "special characters" like `<cr>`. There are a number of ways around this, but the easiest to use and read is `execute`.

When `execute` looks at the string you tell it to run, it will substitute any special characters it finds *before* running it. In this case, `\r` is an escape sequence that means "carriage return". The double backslash is also an escape sequence that puts a literal backslash in the string.

If we perform this replacement in our mapping and look at the result we can see that the mapping is going to perform:

```
:normal! ?^==\+$<cr>:nohlsearch<cr>kvg_
                ^^^^           ^^^^
                 ||             ||
These are ACTUAL carriage returns, NOT the four characters
"left angle bracket", "c", "r", and "right angle bracket".
```

So now `normal!` will execute these characters as if we had typed them in normal mode. Let's split them apart at the returns to find out what they're doing:

```
?^==\+$
:nohlsearch
kvg_
```

The first piece, `?^==\+$` performs a search backwards for any line that consists of two or more equal signs and nothing else. This will leave our cursor on the first character of the line of equal signs.

We're searching backwards because when you say "change inside heading" while your cursor is in a section of text, you probably want to change the heading for *that* section, not the next one.

The second piece is the `:nohlsearch` command. This simply clears the search highlighting from the search we just performed so it's not distracting.

The final piece is a sequence of three normal mode commands:

- `k`: move up a line. Since we were on the first character of the line of equal signs, we're now on the first character of the heading text.
- `v`: enter (characterwise) visual mode.
- `g_`: move to the last non-blank character of the current line. We use this instead of `$` because `$` would highlight the newline character as well, and this isn't what we want.

16.3 Results

That was a lot of work, but now we've looked at each part of the mapping. To recap:

- We created a operator-pending mapping for "inside this section's heading".
- We used `execute` and `normal!` to run the normal commands we needed to select the heading, and allowing us to use special characters in those.
- Our mapping searches for the line of equal signs which denotes a heading and visually selects the heading text above that.
- Vim handles the rest.

Let's look at one more mapping before we move on. Run the following command:

```
:onoremap ah :<c-u>execute "normal! ?^==\\+\r:nohlsearch\rg_vk0"<cr>
```

Try it by putting your cursor in a section's text and typing `cah`. This time Vim will delete not only the heading's text but also the line of equal signs that denotes a heading. You can think of this movement as "*around* this section's heading".

What's different about this mapping? Let's look at them side by side:

```
:onoremap ih :<c-u>execute "normal! ?^==\\+$\r:nohlsearch\rkvg_"<cr>
:onoremap ah :<c-u>execute "normal! ?^==\\+$\r:nohlsearch\rg_vk0"<cr>
```

The only difference from the previous mapping is the very end, where we select the text to operate on:

```
inside heading: kvg_
around heading: g_vk0
```

The rest of the mapping is the same, so we still start on the first character of the line of equal signs. From there:

- `g_`: move to the last non-blank character in the line.
- `v`: enter (characterwise) visual mode.
- `k`: move up a line. This puts us on the line containing the heading's text.
- `0`: move to the first character of the line.

The result is that both the text and the equal signs end up visually selected, and Vim performs the operation on both.

16.4 Exercises

Markdown can also have headings delimited with lines of `-` characters. Adjust the regex in these mappings to work for either type of heading. You may want to check out `:help pattern-overview`. Remember that the regex is inside of a string, so backslashes will need to be escaped.

Add two autocommands to your `~/.vimrc` file that will create these mappings. Make sure to only map them in the appropriate buffers, and make sure to group them so they don't get duplicated each time you source the file.

Read `:help normal`.

Read `:help execute`.

Read `:help expr-quote` to see the escape sequences you can use in strings.

Create a "inside next email address" operator-pending mapping so you can say "change inside next email address". `in@` is a good candidate for the keys to map. You'll probably want to use `/...some regex...<cr>` for this.

17 Status Lines

Vim allows you to customize the text in the status line at the bottom of each window. This is done through the `statusline` option. Run the following command:

```
:set statusline=%f
```

You should see the path to the file (relative to the current directory) in the status line. Now run this command:

```
:set statusline=%f\ -\ FileType:\ %y
```

Now you'll see something like `foo.markdown - FileType: [markdown]` in the status line.

If you're familiar with C's `printf` or Python's string interpolation the format of this option may look familiar. If not, the only trick is that things that start with `%` are expanded to different text depending on what comes after them. In our example `%f` is replaced with the filename and `%y` is replaced with the type of the file.

Notice how the spaces in the status line need to be escaped with backslashes. This is because `set` allows you to set multiple options at once, as we saw in the second chapter.

Status lines can get extremely complicated very quickly, so there's a better way to set them that will let us be more clear. Run the following commands:

```
:set statusline=%f          " Path to the file
:set statusline+=\ -\       " Separator
:set statusline+=FileType: " Label
:set statusline+=%y         " Filetype of the file
```

In the first command we used = to wipe out any existing value present. In the rest we used += to build up the option one piece at a time. We also added comments explaining each piece for other people reading the code (or ourselves several months later).

Run the following commands:

```
:set statusline=%l     " Current line
:set statusline+=/     " Separator
:set statusline+=%L    " Total lines
```

Now the status line contains only the current line number and number of lines in the file, and looks something like `12/223`.

17.1 Width and Padding

Additional characters can be used in some of the various `%` codes to change how the information is displayed. Run the following command:

```
:set statusline=[%4l]
```

The line number in the status line will now be preceded by enough spaces to make it at least four characters wide (for example: `[  12]`). This can be useful to prevent the text in the status line from shifting around distractingly.

By default the padding spaces are added on the left side of the value. Run this command:

```
:set statusline=Current:\ %4l\ Total:\ %4L
```

Your status line will now look like this:

```
Current:   12 Total:  223
```

You can use `-` to place padding on the right instead of the left. Run this command:

```
:set statusline=Current:\ %-4l\ Total:\ %-4L
```

Your status line will now look like this:

```
Current: 12   Total: 223
```

This looks much nicer because the numbers are next to their labels.

For codes that result in a number you can tell Vim to pad with zeros instead of spaces. Run the following command:

```
:set statusline=%04l
```

Now your status line will read `0012` when on line twelve.

Finally, you can also set the maximum width of a code's output. Run this command:

```
:set statusline=%F
```

`%F` displays the *full* path to the current file. Now run this command to change the maximum width:

```
:set statusline=%.20F
```

The path will be truncated if necessary, looking something like this:

```
<hapters/17.markdown
```

This can be useful for preventing paths and other long codes from taking up the entire line.

17.2 General Format

The general format for a code in a status line is shown in `:help statusline`:

```
%-0{minwid}.{maxwid}{item}
```

Everything except the `%` and the item is optional.

17.3 Splitting

We're not going to cover status lines in too much detail here (Vim's own documentation on them is very extensive if you want to learn more), but there's one more simple code that can be very useful immediately. Run the following commands:

```
:set statusline=%f          " Path to the file
:set statusline+=%=         " Switch to the right side
:set statusline+=%l         " Current line
:set statusline+=/          " Separator
:set statusline+=%L         " Total lines
```

Now the status line will contain the path to the file on the left side, and the current/total lines on the right side. The `%=` code tells Vim that everything coming after that should be aligned (as a whole) to the right instead of the left.

17.4 Exercises

Skim the list of available codes in `:help statusline`. Don't worry if you don't understand some of them just yet.

Add some lines to your `~/.vimrc` file to build yourself a custom status line. Be sure to use the `+=` form of `set` to split the definition across multiple lines, and add a comment on each line to document what each piece does.

Try using autocommands and `setlocal` to define different status lines for different filetypes. Make sure to wrap the autocommands in groups to prevent duplication (as always).

18 Responsible Coding

So far we've covered a bunch of Vim commands that let you customize Vim quickly. All of them except for autocommand groups were single-line commands that you can add to your ~/.vimrc file in seconds.

In the next part of the book we're going to dive into Vimscript as a real programming language, but before we do that I want to talk a bit about how to stay sane while writing large amounts of Vimscript.

18.1 Commenting

Vimscript is extremely powerful, but has grown organically over the years into a twisty maze ready to ensnare unwary programmers who enter it.

Options and commands are often terse and hard to read, and working around compatibility issues can increase the complexity of your code. Writing a plugin and allowing for user customization introduces another entire layer above that!

Be defensive when writing anything that takes more than a few lines of Vimscript. Add a comment explaining what it does, and if there is a relevant help topic, mention it in the comment!

This not only benefits you when you try to maintain it months or years later, but also helps other people understand it if you share your ~/.vimrc file on Bitbucket or GitHub (which I highly recommend).

18.2 Grouping

Our mappings for editing and sourcing ~/.vimrc have made it quick and easy to add new things to it on the fly. Unfortunately this also makes it easy for it to grow out of control and become hard to navigate.

One way to combat this is to use Vim's code folding capabilities and group lines into sections. If you've never used Vim's folding you should look into it as soon as you can. Some people (myself included) find it indispensable in our day to day coding.

First we need to set up folding for Vimscript files. Add the following lines to your ~/.vimrc file:

```
augroup filetype_vim
    autocmd!
    autocmd FileType vim setlocal foldmethod=marker
augroup END
```

This will tell Vim to use the `marker` method of folding for any Vimscript files.

Go ahead and run `:setlocal foldmethod=marker` in the window with your `~/.vimrc` file now. Sourcing the file won't work, because Vim has already set the FileType for this file and the autocommand only fires when that happens. In the future you won't need to do it manually.

Now add lines before and after that autocommand group so that it looks like this:

```
" Vimscript file settings ---------------------- {{{
augroup filetype_vim
    autocmd!
    autocmd FileType vim setlocal foldmethod=marker
augroup END
" }}}
```

Return to normal mode, put your cursor on any of those lines and type `za`. Vim will fold the lines starting at the one containing `{{{` and ending at the one containing `}}}`. Typing `za` again will unfold the lines.

You may think that adding explicit comments to source code that describe folding is ugly at first. I thought the same way when I first saw it. For most files I still think it's wrong. Not everyone uses the same editor, so littering your code with folding comments is just noisy to anyone else looking at the code in something other than Vim.

Vimscript files are special case, though. It's highly unlikely that someone who doesn't use Vim will be reading your code, and it's especially important to group things explicitly and thoughtfully when writing Vimscript so you don't go crazy.

Try these explicit folds out for a while. You might grow to love them.

18.3 Short Names

Vim allows you to use abbreviated names for most commands and options. For example, both of these commands do exactly the same thing:

```
:setlocal wrap
:setl wrap
```

I'd like to *strongly* caution you against using these abbreviations in your `~/.vimrc` file and in plugins that you write. Vimscript is terse and cryptic enough to begin with; shortening things further is only going to make it even harder to read. Even if *you* know what a certain short command means, someone else reading your code might not.

With that said, the abbreviated forms are *great* for running commands manually in the middle of coding. No one will ever see them again after you press return, so there's no reason to press more keys than you have to.

18.4 Exercises

Go through your entire `~/.vimrc` file and arrange the lines into related groups. Some places to start might be: "Basic Settings", "FileType-specific settings", "Mappings", and "Status Line". Add folding markers with headings to each section.

Find out how to make Vim fold everything automatically the first time you open the file. Look at `:help foldlevelstart` for a good place to start.

Go through your `~/.vimrc` file and change any abbreviated commands and options to their full names.

Look through your `~/.vimrc` file and make sure you don't have any sensitive information inside. Create a git or Mercurial repository, move the file into it, and symlink that file to `~/.vimrc`.

Commit the repository you just made and put it on Bitbucket or GitHub so other people can see it and get ideas for their own. Be sure to commit and push the repository fairly often so your changes are recorded.

If you use Vim on more than one machine, clone down that repository and symlink the file there as well. This will make it simple and easy to use the exact same Vim configuration on all machines you work with.

19 Variables

Up to this point we've covered single commands. For the next third of the book we're going to look at Vimscript as a *programming language*. This won't be as instantly gratifying as the rest of what you've learned, but it will lay the groundwork for the last part of the book, which walks through creating a full-fledged Vim plugin from scratch.

Let's get started. The first thing we need to talk about are variables. Run the following commands:

```
:let foo = "bar"
:echo foo
```

Vim will display `bar`. `foo` is now a variable, and we've assigned it a string: `"bar"`. Now run these commands:

```
:let foo = 42
:echo foo
```

Vim will display `42`, because we've reassigned `foo` to the integer `42`.

From these short examples it may seem like Vimscript is dynamically typed. That's not the case, but we'll talk more about that later.

19.1 Options as Variables

You can read and set *options* as variables by using a special syntax. Run the following commands:

```
:set textwidth=80
:echo &textwidth
```

Vim will display `80`. Using an ampersand in front of a name tells Vim that you're referring to the option, not a variable that happens to have the same name.

Let's see how Vim works with boolean options. Run the following commands:

```
:set nowrap
:echo &wrap
```

Vim displays `0`. Now try these commands:

```
:set wrap
:echo &wrap
```

This time Vim displays `1`. This is a very strong hint that Vim treats the integer `0` as "false" and the integer `1` as "true". It would be reasonable to assume that Vim treats *any* non-zero integer as "truthy", and this is indeed the case.

We can also *set* options as variables using the `let` command. Run the following commands:

```
:let &textwidth = 100
:set textwidth?
```

Vim will display `textwidth=100`.

Why would we want to do this when we could just use `set`? Run the following commands:

```
:let &textwidth = &textwidth + 10
:set textwidth?
```

This time Vim displays `textwidth=110`. When you set an option using `set` you can only set it to a single literal value. When you use `let` and set it as a variable you can use the full power of Vimscript to determine the value.

19.2 Local Options

If you want to set the *local* value of an option as a variable, instead of the *global* value, you need to prefix the variable name.

Open two files in separate splits. Run the following command:

```
:let &l:number = 1
```

Now switch to the other file and run this command:

```
:let &l:number = 0
```

Notice that the first window has line numbers and the second does not.

19.3 Registers as Variables

You can also read and set *registers* as variables. Run the following command:

```
:let @a = "hello!"
```

Now put your cursor somewhere in your text and type `"ap`. This command tells Vim to "paste the contents of register `a` here". We just set the contents of that register, so Vim pastes `hello!` into your text.

Registers can also be read. Run the following command:

```
:echo @a
```

Vim will echo `hello!`.

Select a word in your file and yank it with `y`, then run this command:

```
:echo @"
```

Vim will echo the word you just yanked. The `"` register is the "unnamed" register, which is where text you yank without specifying a destination will go.

Perform a search in your file with `/someword`, then run the following command:

```
:echo @/
```

Vim will echo the search pattern you just used. This lets you programmatically read *and modify* the current search pattern, which can be very useful at times.

19.4 Exercises

Go through your `~/.vimrc` file and change some of the `set` and `setlocal` commands to their `let` forms. Remember that boolean options still need to be set to something.

Try setting a boolean option like `wrap` to something other than zero or one. What happens when you set it to a different number? What happens if you set it to a string?

Go back through your `~/.vimrc` file and undo the changes. You should never use `let` if `set` will suffice – it's harder to read.

Read `:help registers` and look over the list of registers you can read and write.

20 Variable Scoping

So far Vimscript's variables may seem familiar if you come from a dynamic language like Python or Ruby. For the most part variables act like you would expect, but Vim adds a certain twist to variables: scoping.

Open two different files in separate splits, then go into one of them and run the following commands:

```
:let b:hello = "world"
:echo b:hello
```

As expected, Vim displays `world`. Now switch to the other buffer and run the `echo` command again:

```
:echo b:hello
```

This time Vim throws an error, saying it can't find the variable.

When we used `b:` in the variable name we told Vim that the variable `hello` should be local to the current buffer.

Vim has many different scopes for variables, but we need to learn a little more about Vimscript before we can take advantage of the rest. For now, just remember that when you see a variable that start with a character and a colon that it's describing a scoped variable.

20.1 Exercises

Skim over the list of scopes in `:help internal-variables`. Don't worry if you don't know what some of them mean, just take a look and keep them in the back of your mind.

21 Conditionals

Every programming language has a way to branch, and in Vimscript that method is the `if` statement. The `if` statement is the core of branching in Vim. There's no `unless` statement like Ruby, so any decision making you do in your coding will be done with `if`s.

Before we talk about Vim's `if` statement we need to take a short detour to talk about syntax so we're all on the same page.

21.1 Multiple-Line Statements

Sometimes you can't fit a piece of Vimscript on a single line of code. We saw this when we talked about autocommand groups. Here's a chunk of code we used before:

```
:augroup testgroup
:    autocmd BufWrite * :echom "Baz"
:augroup END
```

You can write this as three separate lines in a Vimscript file, which is ideal, but it gets tedious to write this way when running commands manually. Instead you can separate each line with a pipe character (`|`). Run the following command:

```
:echom "foo" | echom "bar"
```

Vim will treat that as two separate commands. Use `:messages` to check the log if you didn't see both lines appear.

For the rest of this book if you want to manually run a command but don't want to bother typing in the newlines and colons, feel free to put it all on one line separated by pipes.

21.2 Basic If

Now that we've got that out of the way, run the following commands:

```
:if 1
:    echom "ONE"
:endif
```

Vim will display `ONE`, because the integer `1` is "truthy". Now try these commands:

```
:if 0
:    echom "ZERO"
:endif
```

Vim will *not* display `ZERO` because the integer `0` is "falsy". Let's see how strings behave. Run these commands:

```
:if "something"
:    echom "INDEED"
:endif
```

The results may surprise you. Vim does *not* necessarily treat a non-empty string as "truthy", so it will not display anything!

Let's dive a bit further down the rabbit hole. Run these commands:

```
:if "9024"
:    echom "WHAT?!"
:endif
```

This time Vim *does* display the text! What's going on here?

To try to wrap our heads around what's going on, run the following three commands:

```
:echom "hello" + 10
:echom "10hello" + 10
:echom "hello10" + 10
```

The first command causes Vim to echo `10`, the second command echoes `20`, and the third echoes `10` again!

After observing all of these commands we can draw a few informed conclusions about Vimscript:

- Vim will try to coerce variables (and literals) when necessary. When `10 + "20foo"` is evaluated Vim will convert `"20foo"` to an integer (which results in `20`) and then add it to `10`.
- Strings that start with a number are coerced to that number, otherwise they're coerced to `0`.
- Vim will execute the body of an `if` statement when its condition evaluates to a non-zero integer, *after* all coercion takes place.

21.3 Else and Elseif

Vim, like Python, supports both "else" and "else if" clauses. Run the following commands:

```
:if 0
:    echom "if"
:elseif "nope!"
:    echom "elseif"
:else
:    echom "finally!"
:endif
```

Vim echoes `finally!` because both of the previous conditions evaluate to zero, which is falsy.

21.4 Exercises

Drink a beer to console yourself about Vim's coercion of strings to integers.

22 Comparisons

We've gone over conditionals, but `if` statements aren't very useful if we can't compare things. Of course Vim lets us compare values, but it's not as straightforward as it may seem.

Run the following commands:

```
:if 10 > 1
:    echom "foo"
:endif
```

Vim will, of course, display `foo`. Now run these commands:

```
:if 10 > 2001
:    echom "bar"
:endif
```

Vim displays nothing, because `10` is not greater than `2001`. So far everything works as expected. Run these commands:

```
:if 10 == 11
:    echom "first"
:elseif 10 == 10
:    echom "second"
:endif
```

Vim displays `second`. Nothing surprising here. Let's try comparing strings. Run these commands:

```
:if "foo" == "bar"
:    echom "one"
:elseif "foo" == "foo"
:    echom "two"
:endif
```

Vim echoes `two`. There's still nothing surprising, so what was I going on about at the beginning of the chapter?

22.1 Case Sensitivity

Run the following commands:

```
:set noignorecase
:if "foo" == "FOO"
:    echom "vim is case insensitive"
:elseif "foo" == "foo"
:    echom "vim is case sensitive"
:endif
```

Vim evaluates the `elseif`, so apparently Vimscript is case sensitive. Good to know, but nothing earth-shattering. Now run these commands:

```
:set ignorecase
:if "foo" == "FOO"
:    echom "no, it couldn't be"
:elseif "foo" == "foo"
:    echom "this must be the one"
:endif
```

Whoa. Stop right there. Yes, you saw that right.

The behavior of == depends on a user's settings.

I promise I'm not messing with you. Try it again and see. I'm not kidding, I can't make this stuff up.

22.2 Code Defensively

What does this mean? It means that you can *never* trust the == comparison when writing a plugin for other people to use. A bare == should *never* appear in your plugins' code.

This idea is the same as the "`nmap` versus `nnoremap`" one. *Never* trust your users' settings. Vim is old, vast, and complicated. When writing a plugin you *have* to assume that users will have every variation of every setting.

So how can you get around this ridiculousness? It turns out that Vim has *two extra sets* of comparison operators to deal with this.

Run the following commands:

```
:set noignorecase
:if "foo" ==? "FOO"
:    echom "first"
:elseif "foo" ==? "foo"
:    echom "second"
:endif
```

Vim displays `first` because `==?` is the "case-insensitive no matter what the user has set" comparison operator. Now run the following commands:

```
:set ignorecase
:if "foo" ==# "FOO"
:    echom "one"
:elseif "foo" ==# "foo"
:    echom "two"
:endif
```

Vim displays `two` because `==#` is the "case-sensitive no matter what the user has set" comparison operator.

The moral of this story is that you should *always* use explicit case sensitive or insensitive comparisons. Using the normal forms is *wrong* and it *will* break at some point. Save yourself the trouble and type the extra character.

When you're comparing integers this distinction obviously doesn't matter. Still, I feel that it's better to use the case-sensitive comparisons everywhere (even where they're not strictly needed), than to forget them in a place that they *are* needed.

Using `==#` and `==?` with integers will work just fine, and if you change them to strings in the future it will work correctly. If you'd rather use `==` for integers that's fine, just remember that you'll need to change the comparison if you change them to strings in the future.

22.3 Exercises

Play around with `:set ignorecase` and `:set noignorecase` and see how various comparisons act.

Read `:help ignorecase` to see why someone might set that option.

Read `:help expr4` to see all the available comparison operators.

23 Functions

Like most programming languages, Vimscript has functions. Let's take a look at how to create them, and then talk about some of their quirks.

Run the following command:

```
:function meow()
```

You might think this would start defining a function named `meow`. Unfortunately this is not the case, and we've already run into one of Vimscript's quirks.

Vimscript functions *must* start with a capital letter if they are unscoped!

Even if you *do* add a scope to a function (we'll talk about that later) you may as well capitalize the first letter of function names anyway. Most Vimscript coders seem to do it, so don't break the convention.

Okay, let's define a function for real this time. Run the following commands:

```
:function Meow()
:  echom "Meow!"
:endfunction
```

This time Vim will happily define the function. Let's try running it:

```
:call Meow()
```

Vim will display `Meow!` as expected.

Let's try returning a value. Run the following commands:

```
:function GetMeow()
:  return "Meow String!"
:endfunction
```

Now try it out by running this command:

```
:echom GetMeow()
```

Vim will call the function and give the result to `echom`, which will display `Meow String!`.

23.1 Calling Functions

We can already see that there are two different ways of calling functions in Vimscript.

When you want to call a function directly you use the `call` command. Run the following commands:

```
:call Meow()
:call GetMeow()
```

The first will display `Meow!` but the second doesn't display anything. The return value is thrown away when you use `call`, so this is only useful when the function has side effects.

The second way to call functions is in expressions. You don't need to use `call` in this case, you can just name the function. Run the following command:

```
:echom GetMeow()
```

As we saw before, this calls `GetMeow` and passes the return value to `echom`.

23.2 Implicit Returning

Run the following command:

```
:echom Meow()
```

This will display two lines: `Meow!` and `0`. The first obviously comes from the `echom` inside of `Meow`. The second shows us that if a Vimscript function doesn't return a value, it implicitly returns `0`. Let's use this to our advantage. Run the following commands:

```
:function TextwidthIsTooWide()
:  if &l:textwidth ># 80
:    return 1
:  endif
:endfunction
```

This function uses a lot of important concepts we've seen before:

- `if` statements
- Treating options as variables
- Localizing those option variables
- Case-sensitive comparisons

If any of those sound unfamiliar you should go back a few chapters and read about them.

We've now defined a function that will tell us if the `textwidth` setting is "too wide" in the current buffer (because 80 characters is, of course, the correct width for anything but HTML).

Let's try using it. Run the following commands:

```
:set textwidth=80
:if TextwidthIsTooWide()
:  echom "WARNING: Wide text!"
:endif
```

What did we do here?

- First we set the `textwidth` globally to `80`.
- The we ran an if statement that checked if `TextwidthIsTooWide()` was truthy.
- This wound up not being the case, so the `if`'s body wasn't executed.

Because we never explicitly returned a value, Vim returned `0` from the function, which is falsy. Let's try changing that. Run the following commands:

```
:setlocal textwidth=100
:if TextwidthIsTooWide()
:  echom "WARNING: Wide text!"
:endif
```

This time the `if` statement in the function executes its body, returns `1`, and so the `if` we manually typed in executes *its* body.

23.3 Exercises

Read `:help :call`. Ignore anything about "ranges" for now. How many arguments can you pass to a function? Is this surprising?

Read the first paragraph of `:help E124` and find out what characters you're allowed to use in function names. Are underscores okay? Dashes? Accented characters? Unicode characters? If it's not clear from the documentation just try them out and see.

Read `:help return`. What's the "short form" of that command (which I told you to never use)? Is it what you expected? If not, why not?

24 Function Arguments

Vimscript functions can, of course, take arguments. Run the following commands:

```
:function DisplayName(name)
:  echom "Hello!  My name is:"
:  echom a:name
:endfunction
```

Run the function:

```
:call DisplayName("Your Name")
```

Vim will display two lines: `Hello!  My name is:` and `Your Name`.

Notice the `a:` in the name of the variable that we passed to the `echom` command. This represents a variable scope, which we talked about in an earlier chapter.

Let's remove this scope prefix and see how Vim reacts. Run the following commands:

```
:function UnscopedDisplayName(name)
:  echom "Hello!  My name is:"
:  echom name
:endfunction
:call UnscopedDisplayName("Your Name")
```

This time Vim complains that it can't find the variable `name`.

When you write a Vimscript function that takes arguments you *always* need to prefix those arguments with `a:` when you use them to tell Vim that they're in the argument scope.

24.1 Varargs

Vimscript functions can optionally take variable-length argument lists like Javascript and Python. Run the following commands:

```
:function Varg(...)
:  echom a:0
:  echom a:1
:  echo a:000
:endfunction

:call Varg("a", "b")
```

This function shows us several things, so let's look at each one individually.

The `...` in the function definition tells Vim that this function can take any number of arguments. This is like a `*args` argument in a Python function.

The first line of the function echoes the message `a:0` and displays `2`. When you define a function that takes a variable number of arguments in Vim, `a:0` will be set to the number of extra arguments you were given. In this case we passed two arguments to `Varg` so Vim displayed `2`.

The second line echoes `a:1` which displays `a`. You can use `a:1`, `a:2`, etc to refer to each extra argument your function receives. If we had used `a:2` Vim would have displayed "b".

The third line is a bit trickier. When a function has varargs, `a:000` will be set to a list containing all the extra arguments that were passed. We haven't looked at lists quite yet, so don't worry about this too much. You can't use `echom` with a list, which is why we used `echo` instead for that line.

You can use varargs together with regular arguments too. Run the following commands:

```
:function Varg2(foo, ...)
:  echom a:foo
:  echom a:0
:  echom a:1
:  echo a:000
:endfunction

:call Varg2("a", "b", "c")
```

We can see that Vim puts `"a"` into the named argument `a:foo`, and the rest are put into the list of varargs.

24.2 Assignment

Try running the following commands:

```
:function Assign(foo)
:  let a:foo = "Nope"
:  echom a:foo
:endfunction

:call Assign("test")
```

Vim will throw an error, because you can't reassign argument variables. Now run these commands:

```
:function AssignGood(foo)
:  let foo_tmp = a:foo
:  let foo_tmp = "Yep"
:  echom foo_tmp
:endfunction

:call AssignGood("test")
```

This time the function works, and Vim displays `Yep`.

24.3 Exercises

Read the first two paragraphs of `:help function-argument`.

Read `:help local-variables`.

25 Numbers

Now it's time to start taking a closer look at the different types of variables you can use. First we'll go over Vim's numeric types.

Vimscript has two types of numeric variables: Numbers and Floats. A Number is a 32 bit signed integer. A Float is a floating point number.

25.1 Number Formats

You can specify Numbers in a few different ways. Run the following command:

```
:echom 100
```

No surprises here – Vim displays `100`. Now run this command:

```
:echom 0xff
```

This time Vim displays `255`. You can specify numbers in hex notation by prefixing them with `0x` or `0X`. Now run this command:

```
:echom 010
```

You can also use octal by starting a number with a `0`. Be careful with this, because it's easy to make mistakes. Try the following commands:

```
:echom 017
:echom 019
```

Vim will print `15` for the first command, because `17` in octal is equal to `15` in decimal. For the second command Vim treats it as a decimal number, even though it starts with a `0`, because it's not a valid octal number.

Because Vim silently does the wrong thing in this case, I'd recommend avoiding the use of octal numbers when possible.

25.2 Float Formats

Floats can also be specified in multiple ways. Run the following command:

```
:echo 100.1
```

Notice that we're using `echo` here and not `echom` like we usually to. We'll talk about why in a moment.

Vim displays `100.1` as expected. You can also use exponential notation. Run this command:

```
:echo 5.45e+3
```

Vim displays `5450.0`. A negative exponent can also be used. Run this command:

```
:echo 15.45e-2
```

Vim displays `0.1545`. The `+` or `-` before the power of ten is optional. If it's omitted then it's assumed to be positive. Run the following command:

```
:echo 15.3e9
```

Vim will display `1.53e10`, which is equivalent. The decimal point and number after it are *not* optional. Run the following command and see that it crashes:

```
:echo 5e10
```

25.3 Coercion

When you combine a Number and a Float through arithmetic, comparison, or any other operation Vim will cast the Number to a Float, resulting in a Float. Run the following command:

```
:echo 2 * 2.0
```

Vim displays `4.0`.

25.4 Division

When dividing two Numbers, the remainder is dropped. Run the following command:

```
:echo 3 / 2
```

Vim displays `1`. If you want Vim to perform floating point division one of the numbers needs to be a Float, which will cause the other one to be coerced to a Float as well. Run this command:

```
:echo 3 / 2.0
```

Vim displays `1.5`. The `3` is coerced to a Float, and then normal floating point division is performed.

25.5 Exercises

Read `:help Float`. When might floating point number not work in Vimscript?

Read `:help floating-point-precision`. What might this mean if you're writing a Vim plugin that deals with floating point numbers?

26 Strings

The next type of variable we'll look at is the String. Since Vim is all about manipulating text you'll be using this one quite a bit.

Run the following command:

```
:echom "Hello"
```

Vim will echo `Hello`. So far, so good.

26.1 Concatenation

One of the most common things you'll want to do with strings is adding them together. Run this command:

```
:echom "Hello, " + "world"
```

What happened? Vim displayed `0` for some reason!

Here's the issue: Vim's `+` operator is *only* for Numbers. When you pass a string to `+` Vim will try to coerce it to a Number before performing the addition. Run the following command:

```
:echom "3 mice" + "2 cats"
```

This time Vim displays `5`, because the strings are coerced to the numbers `3` and `2` respectively.

When I said "Number" I really *meant* Number. Vim will *not* coerce strings to Floats! Try this command to see prove this:

```
:echom 10 + "10.10"
```

Vim displays `20` because it dropped everything after the decimal point when coercing `10.10` to a Number.

To combine strings you need to use the concatenation operator. Run the following command:

```
:echom "Hello, " . "world"
```

This time Vim displays `Hello, world`. `.` is the "concatenate strings" operator in Vim, which lets you combine strings. It doesn't add whitespace or anything else in between.

Coercion works both ways. Kind of. Try this command:

```
:echom 10 . "foo"
```

Vim will display `10foo`. First it coerces `10` to a String, then it concatenates it with the string on the right hand side. Things get a bit stickier when we're working with Floats, though. Run this command:

```
:echom 10.1 . "foo"
```

This time Vim throws an error, saying we're using a Float as a String. Vim will happily let you use a String as a Float when performing addition, but *won't* let you use a Float as a String when concatenating.

The moral of this story is that Vim is a lot like Javascript: it allows you to play fast and loose with types sometimes, but it's a really bad idea to do so because it will come back to bite you at some point.

When writing Vimscript, make sure you know what the type of each of your variables is. If you need to change that type you should use a function to explicitly change it, even if it's not strictly necessary at the moment. Don't rely on Vim's coercion because at some point you *will* regret it.

26.2 Special Characters

Like most programming languages, Vimscript lets you use escape sequences in strings to represent hard-to-type characters. Run the following command:

```
:echom "foo \"bar\""
```

The `\"` in the string is replaced with a double quote character, as you would probably expect. Escape sequences work mostly as you would expect. Run the following command:

```
:echom "foo\\bar"
```

Vim displays `foo\bar`, because `\\` is the escape sequence for a literal backslash, just like in most programming languages. Now run the following command (note that it's an `echo` and *not* an `echom`):

```
:echo "foo\nbar"
```

This time Vim will display two lines, `foo` and `bar`, because the `\n` is replaced with a newline. Now try running this command:

```
:echom "foo\nbar"
```

Vim will display something like `foo^@bar`. When you use `echom` instead of `echo` with a String Vim will echo the *exact* characters of the string, which sometimes means that it will show a different representation than plain old `echo`. `^@` is Vim's way of saying "newline character".

26.3 Literal Strings

Vim also lets you use "literal strings" to avoid excessive use of escape sequences. Run the following command:

```
:echom '\n\\'
```

Vim displays `\n\\`. Using single quotes tells Vim that you want the string *exactly* as-is, with no escape sequences. The one exception is that two single quotes in a row will produce one single quote. Try this command:

```
:echom 'That''s enough.'
```

Vim will display `That's enough.`. Two single quotes is the *only* sequence that has special meaning in a literal string.

We'll revisit literal strings when they become most useful, later in the book (when we dive into regular expressions).

26.4 Truthiness

You might be wondering how Vim treats strings when used in an `if` statement. Run the following command:

```
:if "foo"
:  echo "yes"
:else
:  echo "no"
:endif
```

Vim will display `no`. If you're wondering why this happens you should reread the chapter on conditionals, because we talked about it there.

26.5 Exercises

Read `:help expr-quote`. Review the list of escape sequences you can use in a normal Vim string. Find out how to insert a tab character.

Try to figure out a way to insert a tab character into a string *without* using an escape sequence. Read `:help i_CTRL-V` for a hint.

Read `:help literal-string`.

27 String Functions

Vim has many built-in functions to manipulate strings. In this chapter we'll look at a few of the most important ones.

27.1 Length

The first function we'll look at is `strlen`. Run the following command:

```
:echom strlen("foo")
```

Vim displays `3`, which is the length of the string `"foo"`. Now try the following command:

```
:echom len("foo")
```

Vim once again displays `3`. When used with Strings `len` and `strlen` have identical effects. We'll come back to `len` later in the book.

27.2 Splitting

Run the following command (note that it's an `echo` and not an `echom`):

```
:echo split("one two three")
```

Vim displays `['one', 'two', 'three']`. The `split` function splits a String into a List. We'll talk about Lists shortly, but for now don't worry too much about them.

You can also tell Vim to use a separator other than "whitespace" for splitting. Run the following command:

```
:echo split("one,two,three", ",")
```

Vim will once again display `['one', 'two', 'three']`, because the second argument to `split` tells it to split the string on the comma character instead of on whitespace.

27.3 Joining

Not only can you split strings, you can also join them. Run the following command:

```
:echo join(["foo", "bar"], "...")
```

Vim will display `foo...bar`. Don't worry about the list syntax for now.

`split` and `join` can be paired to great effect. Run the following command:

```
:echo join(split("foo bar"), ";")
```

Vim displays `foo;bar`. First we split the string `"foo bar"` into a list, then we joined that list together using a semicolon as the separator.

27.4 Lower and Upper Case

Vim has two functions to change the case of Strings. Run the following commands:

```
:echom tolower("Foo")
:echom toupper("Foo")
```

Vim displays `foo` and `FOO`. This should be pretty easy to understand.

In many languages (like Python) a common idiom is to force strings to lowercase before comparing them to perform a case-insensitive comparison. In Vimscript this isn't necessary because we have the case-insensitive comparison operators. Reread the chapter on comparisons if you don't remember those.

It's up to you to decide whether to use `tolower` and `==#`, or just `==?` to perform case-sensitive comparisons. There doesn't seem to be any strong preference in the Vimscript community. Pick one and stick to it for all of your scripts.

27.5 Exercises

Run `:echo split('1 2')` and `:echo split('1,,2', ',')`. Do they behave the same?

Read `:help split()`.

Read `:help join()`.

Read `:help functions` and skim the list of built-in functions for ones that mention the word "String". Use the `/` command to make it easier (remember, Vim's help files can be navigated like any other kind of file). There are a *lot* of functions here, so don't feel like you need to read the documentation for every single one. Just try to get an idea of what's available if you need it in the future.

28 Execute

The `execute` command is used to evaluate a string as if it were a Vimscript command. We saw it in an earlier chapter, but now that we know a bit more about Vimscript Strings we're going to take another look.

28.1 Basic Execution

Run the following command:

```
:execute "echom 'Hello, world!'"
```

Vim evaluates `echom 'Hello, world!'` as a command and dutifully echoes it to the screen and message log. Execute is a very powerful tool because it lets you build commands out of arbitrary strings.

Let's try a more useful example. Prepare by opening a file in Vim, then using `:edit foo.txt` in the same window to open a new buffer. Now run the following command:

```
:execute "rightbelow vsplit " . bufname("#")
```

Vim will open the first file in a vertical split to the right of the second file. What happened here?

First, Vim builds the command string by concatenating `"rightbelow vsplit "` with the result of the `bufname("#")` call.

We'll look at the function more later, but for now just trust that it returns the path of the previous buffer. You can play with it using `echom` if you want to see for yourself.

Once `bufname` is evaluated Vim the string `"rightbelow vsplit bar.txt"`. The `execute` command evaluates this as a Vimscript command which opens the split with the file.

28.2 Is Execute Dangerous?

In most programming languages the use of such an "eval" construct to evaluate strings as program code is frowned upon (to put it lightly). Vimscript's `execute` command doesn't have the same stigma for two reasons.

First, most Vimscript code only ever takes input from a single person: the user. If the user wants to input a tricky string that will cause an `execute` command to do something bad, well, it's their computer! Contrast this with other languages, where programs constantly take input from untrusted users. Vim is a unique environment where the normal security concerns simply aren't common.

The second reason is that because Vimscript has sometimes arcane and tricky syntax, `execute` is often the easiest, most straightforward way to get something done. In most other languages using an "eval" construct won't usually save you much typing, but in Vimscript it can collapse many lines into a single one.

28.3 Exercises

Skim `:help execute` to get an idea of some of the things you can and can't use `execute` for. Don't dive too deeply yet – we're going to revisit it very soon.

Read `:help leftabove`, `:help rightbelow`, `:help :split`, and `:help :vsplit` (notice the extra colon in the last two topics).

Add a mapping to your `~/.vimrc` file that opens the previous buffer in a split of your choosing (vertical/horizontal, above/below/left/right).

29 Normal

So far we've covered some of the most useful Vimscript commands, but what about all the stuff you do on a daily basis in normal mode? Can we somehow use all the knowledge we have from editing text in our scripting?

The answer is: "of course". We've seen the `normal` command before, and now it's time to revisit it in a bit more detail. Run the following command:

```
:normal G
```

Vim will move your cursor to the last line in the current file, just like pressing `G` in normal mode would. Now run the following command:

```
:normal ggdd
```

Vim will move to the first line in the file (`gg`) and then delete it (`dd`).

The `normal` command simply takes a sequence of keys and pretends they were typed in normal mode. Seems simple enough.

29.1 Avoiding Mappings

Run the following command to map the `G` key to something else:

```
:nnoremap G dd
```

Now pressing `G` in normal mode will delete a line. Try this command:

```
:normal G
```

Vim will delete the current line. The `normal` command will take into account any mappings that exist.

This means that we need something like the `nnoremap` version of `nmap` for `normal`, otherwise we'll never be able to use it since we can't know what keys our users have mapped.

Luckily Vim has a `normal!` command that does exactly this. Run this command:

```
:normal! G
```

This time Vim moves to the bottom of the file even though `G` has been mapped.

When writing Vim scripts you should **always** use `normal!`, and **never** use plain old `normal`. You can't trust what keys your users will have mapped in their `~/.vimrc` files.

29.2 Special Characters

If you play around with `normal!` long enough you'll probably notice a problem. Try the following command:

```
:normal! /foo<cr>
```

At first glance it may seem like this should perform a search for `foo`, but you'll see that it doesn't work. The problem is that `normal!` doesn't parse special character sequences like `<cr>`.

In this case Vim thinks you wanted to search for the character sequence "f, o, o, left angle bracket, c, r, right angle bracket", and doesn't realize that you even pressed return to perform the search! We'll talk about how to get around this in the next chapter.

29.3 Exercises

Read `:help normal`. The end of it will hint at the topic of the next chapter.

29.4 Extra Credit

If you're not feeling up for a challenge, skip this section. If you are, good luck!

Recall what `:help normal` said about undo. Try to make a mapping that will delete two lines but let you undo each deletion separately. `nnoremap <leader>d dddd` is a good place to start.

You won't actually need `normal!` for this (`nnoremap` will suffice), but it illustrates a good point: sometimes reading about one Vim command can spark an interest in something unrelated.

If you've never used the `helpgrep` command you'll probably need it now. Read `:help helpgrep`. Pay attention to the parts about how to navigate between the matches.

Don't worry about patterns yet, we're going to cover them soon. For now it's enough to know that you can use something like `foo.*bar` to find lines containing that regex in the documentation.

Unfortunately `helpgrep` can be frustrating at times because you need to know what words to search for before you can find them! I'll cut you some slack and tell you that in this case you're looking

for a way to break Vim's undo sequence manually, so that the two deletes in your mapping can be undone separately.

In the future, be pragmatic. Sometimes Google is quicker and easier when you don't know exactly what you're after.

30 Execute Normal!

Now that we've seen `execute` and `normal!` we can talk about a common Vimscript idiom in more detail. Run the following command:

```
:execute "normal! gg/foo\<cr>dd"
```

This will move to the top of the file, search for the first occurrence of `foo`, and delete the line that contains it.

Previously we tried to use `normal!` with a search command but couldn't enter the return needed to actually perform the search. Combining `normal!` with `execute` fixes that problem.

`execute` lets you build commands programmatically, so you can use Vim's normal string escape sequences to generate the non-printing characters you need. Try the following command:

```
:execute "normal! mqA;\<esc>`q"
```

What does this do? Let's break it apart:

- `:execute "normal! ..."`: run the sequence of commands as if they were entered in normal mode, ignoring all mappings, and replacing string escape sequences with their results.
- `mq`: store the current location in mark "q".
- `A`: move to the end of the current line and enter insert mode after the last character.
- `;`: we're now in insert mode, so just put a literal semicolon in the file.
- `\<esc>`: this is a string escape sequence which resolves to a press of the escape key, which takes us out of insert mode.
- "`q": return to the exact location of mark "q".

It looks a bit scary but it's actually quite useful: it puts a semicolon at the end of the current line while leaving your cursor in place. A mapping for this could come in handy if you forget a semicolon when writing Javascript, C, or any other language with semicolons as statement terminators.

30.1 Exercises

Read `:help expr-quote` again (you've seen it before) to remind yourself how to use string escapes to pass special characters to `normal!` with `execute`.

Put down this book for a while before you go on to the next chapter. Get a sandwich or a cup of coffee, feed your pets if you have them, and relax for a bit before continuing.

31 Basic Regular Expressions

Vim is a text editor, which means that a great deal of your Vimscript code will be dedicated to working with text. Vim has powerful support for regular expressions, but as usual there are some quirks.

Type the following text into a buffer:

```
max = 10

print "Starting"

for i in range(max):
    print "Counter:", i

print "Done"
```

This is the text we'll use to experiment with Vimscript's regex support. It happens to be Python code, but don't worry if you don't know Python. It's just an example.

I'm going to assume that you know the basics of regular expressions. If you don't you should stop reading this book and start reading Learn Regex the Hard Way[1] by Zed Shaw. Come back when you're done with that.

31.1 Highlighting

Before we start we need to turn on search highlighting so we can see what we're doing. Run the following command:

```
:set hlsearch incsearch
```

`hlsearch` tells Vim to highlight all matches in a file when you perform a search, and `incsearch` tells Vim to highlight the *next* match while you're still typing out your search pattern.

31.2 Searching

Put your cursor at the top of the file and run the following command:

[1] http://regex.learncodethehardway.org/

```
/print
```

As you type in each letter, Vim will start highlighting them in the first line. When you press return to execute the search *all* the instances of `print` will be highlighted and your cursor will be moved to the next match.

Now try running the following command:

```
:execute "normal! gg/print\<cr>"
```

This will go to the top of the file and perform a search for `print`, putting us at the first match. It does this using `:execute "normal! ..."` which we saw in the previous chapter.

To get to the second match in the file you can just add more commands onto the end of the string. Run this command:

```
:execute "normal! gg/print\<cr>n"
```

Vim will put the cursor on the second `print` in the buffer (and all the matches will be highlighted).

Let's try going in the opposite direction. Run this command:

```
:execute "normal! G?print\<cr>"
```

This time we move to the bottom of the file with `G` and use `?` to search backward instead of forward.

All of these searching commands should be familiar – we're mostly going over them to get you used to the `:execute "normal! ..."` idiom, because it will let you do anything you know how to do in vanilla Vim in your Vimscript code.

31.3 Magic

The `/` and `?` commands actually take regular expressions, not just literal characters. Run the following command:

```
:execute "normal! gg/for .+ in .+:\<cr>"
```

Vim complains that the pattern is not found! I told you that Vim supports regular expressions in searches, so what's going on? Try the following command:

```
:execute "normal! gg/for .\\+ in .\\+:\<cr>"
```

This time Vim highlights the "for" loop as we expected in the first place. Take a minute and try to think about what exactly changed before moving on. Remember that `execute` takes a String.

The answer is that there are two reasons we needed to write the command like we did:

- First, `execute` takes a String, so the double backslashes we used turn into single backslashes by the time they get to `normal!`.
- Vim has *four* different "modes" of parsing regular expressions! The default mode requires a backslash before the `+` character to make it mean "1 or more of the preceding character" instead of "a literal plus sign".

You can see this a bit easier by just running the search in Vim directly. Type the following command and press return:

```
/print .\+
```

You can see the `\+` working its magic now. The double backslashes were only used because we were passing the pattern as a String to `execute`.

31.4 Literal Strings

As we mentioned in the chapter on Strings, Vim allows you to use single quotes to define a "literal string" that passes through characters directly. For example, the string `'a\nb'` is four characters long.

Can we use literal strings to avoid having to type those double backslashes? Think about this for a minute or two before you move on, because the answer is a bit more complicated that you might think.

Try running the following command (note the single quotes and single backslashes this time):

```
:execute 'normal! gg/for .\+ in .\+:\<cr>'
```

Vim moves you to the top of the file but doesn't move you to the first match. Is this what you expected?

The command doesn't work because we need the `\<cr>` in the pattern to be escaped into a real carriage return character, which tells the search command to actually run. Because we're in a literal string, it's the equivalent of typing `/for .\+ in .\+:\<cr>` in vanilla Vim, which obviously isn't what we want.

All hope is not lost, though! Remember that Vim allows you to concatenate strings, so for larger commands we can use this to split apart the string into easier to read chunks. Run the following command:

```
:execute "normal! gg" . '/for .\+ in .\+:' . "\<cr>"
```

This concatenates the three smaller strings before sending them to `execute`, and lets us use a literal string for the regex while using normal strings for everything else.

31.5 Very Magic

You may be wondering about Vimscript's four different modes of regex parsing and how they're different from the regular expressions you're used to from languages like Python, Perl or Ruby. You can read their documentation if you really want to, but if you want the sane, easy solution just read on.

Run the following command:

```
:execute "normal! gg" . '/\vfor .+ in .+:' . "\<cr>"
```

We've split the pattern out from the rest of the command into its own literal string again, and this time we started the pattern with `\v`. This tells Vim to use its "very magic" regex parsing mode, which is pretty much the same as you're used to in any other programming language.

If you simply start all of your regular expressions with `\v` you'll never need to worry about Vimscript's three other crazy regex modes.

31.6 Exercises

Read `:help magic` carefully.

Read `:help pattern-overview` to see the kinds of things Vim regexes support. Stop reading after the character classes.

Read `:help match`. Try running the `:match Error /\v.../` command a few times by hand.

Edit your `~/.vimrc` file to add a mapping that will use `match` to highlight trailing whitespace as an error. A good key to use might be `<leader>w`.

Add another mapping that will clear the match (perhaps `<leader>W`).

Add a normal mode mapping that will automatically insert the `\v` for you whenever you begin a search. If you're stuck remember that Vim's mappings are extremely simple and you just need to tell it which keys to press when you use the mapped key.

Add the `hlsearch` and `incsearch` options to your `~/.vimrc` file, set however you prefer.

Read `:help nohlsearch`. Note that this is a *command* and *not* the "off mode" setting of `hlsearch`!

Add a mapping to "stop highlighting items from the last search" to your `~/.vimrc` file.

32 Case Study: Grep Operator, Part One

In this chapter and the next we're going to walk through creating a fairly complicated piece of Vimscript. We'll talk about several things we haven't seen before, as well as how some of the things we've studied fit together in practice.

As you work through this case study make sure to look up anything unfamiliar with `:help`. If you coast through without fully understanding everything, you won't learn much.

32.1 Grep

If you've never used `:grep` you should take a minute to read `:help :grep` and `:help :make` now. Read `:help quickfix-window` if you've never used the quickfix window before.

In a nutshell: `:grep ...` will run an external grep program with any arguments you give, parse the result, and fill the quickfix list so you can jump to results inside Vim.

Our example is going to make `:grep` easier to invoke by adding a "grep operator" you can use with any of Vim's built-in (or custom!) motions to select the text you want to search for.

32.2 Usage

The first thing you should think about when creating any non-trivial piece of Vimscript is: "how will this functionality be used?". Try to come up with a smooth, easy, intuitive way to invoke it.

In this case I'll do that step for you:

- We're going to create a "grep operator" and bind it to `<leader>g`.
- It will act like any other Vim operator and take a motion (like `w` or `i{`).
- It will perform the search immediately and open the quickfix window to show the results.
- It will *not* jump to the first result, because that can be jarring if the first result isn't what you're expecting.

Some examples of how you might end up using it:

- `<leader>giw`: Grep for the word under the cursor.
- `<leader>giW`: Grep for the WORD under the cursor.
- `<leader>gi'`: Grep for the contents of the single-quoted string you're currently in.

- `viwe<leader>g`: Visually select a word, extend the selection to the end of the word after it, then grep for the selected text.

There are many, *many* other ways to use this. It may seem like it will take a lot of coding, but actually all we need to do is implement the "operator" functionality and Vim will handle the rest.

32.3 A Preliminary Sketch

One thing that's sometimes helpful when writing tricky bits of Vimscript is to simplify your goal and implement *that* to get an idea of the "shape" your final solution will take.

Let's simplify our goal to: "create a mapping to search for the word under the cursor". This is useful but should be easier, so we can get something running much faster. We'll map this to `<leader>g` for now.

We'll start with a skeleton of the mapping and fill it in as we go. Run this command:

```
:nnoremap <leader>g :grep -R something .<cr>
```

If you've read `:help grep` this should be pretty easy to understand. We've looked at lots of mappings before, and there's nothing new here.

Obviously we're not done yet, so let's refine this mapping until it meets our simplified goal.

32.4 The Search Term

First we need to search for the word under the cursor, not the string `something`. Run the following command:

```
:nnoremap <leader>g :grep -R <cword> .<cr>
```

Now try it out. `<cword>` is a special bit of text you can use in Vim's command-line mode, and Vim will replace it with "the word under the cursor" before running the command.

You can use `<cWORD>` to get a WORD instead of a word. Run this command:

```
:nnoremap <leader>g :grep -R <cWORD> .<cr>
```

Now try the mapping when your cursor is over something like `foo-bar`. Vim will grep for `foo-bar` instead of just part of the word.

There's still a problem with our search term: if there are any special shell characters in it Vim will happily pass them along to the external grep command, which will explode (or worse: do something terrible).

Go ahead and try this to make sure it breaks. Type `foo;ls` into a file and run the mapping while your cursor is over it. The grep command will fail, and Vim will actually run an `ls` command as well! Clearly this could be bad if the word contained a command more dangerous than `ls`.

To try to fix this we'll quote the argument in the grep call. Run this command:

```
:nnoremap <leader>g :grep -R '<cWORD>' .<cr>
```

Most shells treat single-quoted text as (almost) literal, so our mapping is much more robust now.

32.5 Escaping Shell Command Arguments

There's still one more problem with the search term. Try the mapping on the word `that's`. It won't work, because the single quote inside the word interferes with the quotes in the grep command!

To get around this we can use Vim's `shellescape` function. Read `:help escape()` and `:help shellescape()` to see how it works (it's pretty simple).

Because `shellescape()` works on Vim strings, we'll need to dynamically build the command with `execute`. First run the following command to transform the `:grep` mapping into `:execute "..."` form:

```
:nnoremap <leader>g :execute "grep -R '<cWORD>' ."<cr>
```

Try it out and make sure it still works. If not, find any typos and fix them. Then run the following command, which uses `shellescape` to fix the search term:

```
:nnoremap <leader>g :execute "grep -R " . shellescape("<cWORD>") . " ."<cr>
```

Try it out by running it on a normal word like `foo`. It will work properly. Now try it out on a word with a quote in it, like `that's`. It still doesn't work! What happened?

The problem is that Vim performed the `shellescape()` call *before* it expanded out special strings like `<cWORD>` in the command line. So Vim shell-escaped the literal string `"<cWORD>"` (which did nothing but add single quotes to it) and then concatenated it with the strings of our `grep` command.

You can see this by running the following command:

```
:echom shellescape("<cWORD>")
```

Vim will output `'<cWORD>'`. Note that those quotes are actually part of the string. Vim has prepared it for use as a shell command argument.

To fix this we'll use the `expand()` function to force the expansion of `<cWORD>` into the actual string *before* it gets passed to `shellescape`.

Let's break this apart and see how it works, in steps. Put your cursor over a word with a quote, like `that's`, and run the following command:

```
:echom expand("<cWORD>")
```

Vim outputs `that's` because `expand("<cWORD>")` will return the current word under the cursor as a Vim string. Now let's add `shellescape` back in:

```
:echom shellescape(expand("<cWORD>"))
```

This time Vim outputs `'that'\''s'`. If this looks a little funny, you probably haven't had the pleasure of wrapping your brain around shell-quoting in all its insane glory. For now, don't worry about it. Just trust the Vim has taken the string from `expand` and escaped it properly.

Now that we know how to get a fully-escaped version of the word under the cursor, it's time to concatenate it into our mapping! Run the following command:

```
:nnoremap <leader>g :exe "grep -R " . shellescape(expand("<cWORD>")) . " ."\
<cr>
```

Try it out. This mapping won't break if the word we're searching for happens to contain strange characters.

The process of starting with a trivial bit of Vimscript and transforming it little-by-little into something closer to your goal is one you'll find yourself using often.

32.6 Cleanup

There are still a couple of small things to take care of before our mapping is finished. First, we said that we don't want to go to the first result automatically, and we can use `grep!` instead of plain `grep` to do that. Run this command:

```
:nnoremap <leader>g :execute "grep! -R " . shellescape(expand("<cWORD>")) .\
 " ."<cr>
```

Try it out again and nothing will seem to happen. Vim has filled the quickfix window with the results, but we haven't opened it yet. Run the following command:

```
:nnoremap <leader>g :execute "grep! -R " . shellescape(expand("<cWORD>")) .\
 " ."<cr>:copen<cr>
```

Now try the mapping and you'll see that Vim automatically opens the quickfix window with the search results. All we did was tack `:copen<cr>` onto the end of our mapping.

As the finishing touch we'll remove all the grep output Vim displays while searching. Run the following command:

```
:nnoremap <leader>g :silent execute "grep! -R " . shellescape(expand("<cWOR\
D>")) . " ."<cr>:copen<cr>
```

We're done, so try it out and admire your hard work! The `silent` command just runs the command that follows it while hiding any messages it would normally display.

32.7 Exercises

Add the mapping we just created to your `~/.vimrc` file.

Read `:help :grep` if you didn't read it before.

Read `:help cword`.

Read `:help cnext` and `:help cprevious`. Try them out after using your new grep mapping.

Set up mappings for `:cnext` and `:cprevious` to make it easier to quickly run through matches.

Read `:help expand`.

Read `:help copen`.

Add a height to the `:copen` command in the mapping we created to make sure the quickfix window is opened to whatever height you prefer.

Read `:help silent`.

33 Case Study: Grep Operator, Part Two

Now that we've got a preliminary sketch of our solution, it's time to flesh it out into something powerful.

Remember: our original goal was to create a "grep operator". There are a whole bunch of new things we need to cover to do this, but we're going to follow the same process we did in the last chapter: start with something simple and transform it until it does what you need.

Before we start, comment out the mapping we creating the previous chapter from your `~/.vimrc` file – we're going to use the same keystroke for our new operator.

33.1 Create a File

Creating an operator will take a number of commands and typing those out by hand will get tedious very quickly. You could add it to your `~/.vimrc` file, but let's create a separate file just for this operator instead. It's meaty enough to warrant a file of its own.

First, find your Vim `plugin` directory. On Linux or OS X this will be at `~/.vim/plugin`. If you're on Windows it will be inside the `vimfiles` directory in your home directory. (Use the command: `:echo $HOME` in Vim if you're not sure where this is). If this directory doesn't exist, create it.

Inside `plugin/` create a file named `grep-operator.vim`. This is where you'll place the code for this new operator. When you're editing the file you can run `:source %` to reload the code at any time. This file will also be loaded each time you open Vim just like `~/.vimrc`.

Remember that you *must* write the file before you source it for the changes to be seen!

33.2 Skeleton

To create a new Vim operator you'll start with two components: a function and a mapping. Start by adding the following code to `grep-operator.vim`:

```
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@

function! GrepOperator(type)
    echom "Test"
endfunction
```

Write the file and source it with `:source %`. Try it out by pressing `<leader>giw` to say "grep inside word". Vim will echo `Test` *after* accepting the `iw` motion, which means we've laid out the skeleton.

The function is simple and nothing we haven't seen before, but that mapping is a bit more complicated. First we set the `operatorfunc` option to our function, and then we run `g@` which calls this function as an operator. This may seem a bit convoluted, but it's how Vim works.

For now it's okay to consider this mapping to be black magic. You can delve into the detailed documentation later.

33.3 Visual Mode

We've added the operator to normal mode, but we'll want to be able to use it from visual mode as well. Add another mapping below the first:

```
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>
```

Write and source the file. Now visually select something and press `<leader>g`. Nothing happens, but Vim does echo `Test`, so our function is getting called.

We've seen the `<c-u>` in this mapping before but never explained what it did. Try visually selecting some text and pressing `:`. Vim will open a command line as it usually does when `:` is pressed, but it automatically fills in `'<,'>` at the beginning of the line!

Vim is trying to be helpful and inserts this text to make the command you're about to run function on the visually selected range. In this case, however, we don't want the help. We use `<c-u>` to say "delete from the cursor to the beginning of the line", removing the text. This leaves us with a bare `:`, ready for the `call` command.

The `call GrepOperator()` is simply a function call like we've seen before, but the `visualmode()` we're passing as an argument is new. This function is a built-in Vim function that returns a one-character string representing the last type of visual mode used: `"v"` for characterwise, `"V"` for linewise, and a `Ctrl-v` character for blockwise.

33.4 Motion Types

The function we defined takes a `type` argument. We know that when we use the operator from visual mode it will be the result of `visualmode()`, but what about when we run it as an operator from normal mode?

Edit the function body so the file looks like this:

```
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>

function! GrepOperator(type)
    echom a:type
endfunction
```

Source the file, then go ahead and try it out in a variety of ways. Some examples of the output you get are:

- Pressing `viw<leader>g` echoes `v` because we were in characterwise visual mode.
- Pressing `Vjj<leader>g` echoes `V` because we were in linewise visual mode.
- Pressing `<leader>giw` echoes `char` because we used a characterwise motion with the operator.
- Pressing `<leader>gG` echoes `line` because we used a linewise motion with the operator.

Now we know how we can tell the difference between motion types, which will be important when we select the text to search for.

33.5 Copying the Text

Our function is going to need to somehow get access to the text the user wants to search for, and the easiest way to do that is to simply copy it. Edit the function to look like this:

```
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>

function! GrepOperator(type)
    if a:type ==# 'v'
        execute "normal! `<v`>y"
    elseif a:type ==# 'char'
        execute "normal! `[v`]y"
    else
        return
    endif

    echom @@
endfunction
```

Wow. That's a lot of new stuff. Try it out by pressing things like `<leader>giw`, `<leader>g2e` and `vi(<leader>g`. Each time Vim will echo the text that the motion covers, so clearly we're making progress!

Let's break this new code down one step at a time. First we have an `if` statement that checks the `a:type` argument. If the type is `'v'` it was called from characterwise visual mode, so we do something to copy the visually-selected text.

Notice that we use the case-sensitive comparison `==#`. If we used plain `==` and the user has `ignorecase` set it would match `"V"` as well, which is *not* what we want. Code defensively!

The second case of the `if` fires if the operator was called from normal mode using a characterwise motion.

The final case simply returns. We explicitly ignore the cases of linewise/blockwise visual mode and linewise/blockwise motions. Grep doesn't search across lines by default, so having a newline in the search pattern doesn't make any sense!

Each of our two `if` cases runs a `normal!` command that does two things:

- Visually select the range of text we want by:
  - Moving to mark at the beginning of the range.
  - Entering characterwise visual mode.
  - Moving to the mark at the end of the range.
- Yanking the visually selected text.

Don't worry about the specific marks for now. You'll learn why they need to be different when you complete the exercises at the end of this chapter.

The final line of the function echoes the variable `@@`. Remember that variables starting with an `@` are registers. `@@` is the "unnamed" register: the one that Vim places text into when you yank or delete without specify a particular register.

In a nutshell: we select the text to search for, yank it, then echo the yanked text.

33.6 Escaping the Search Term

Now that we've got the text we need in a Vim string we can escape it like we did in the previous chapter. Modify the `echom` command so it looks like this:

```
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>

function! GrepOperator(type)
    if a:type ==# 'v'
        normal! `<v`>y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    echom shellescape(@@)
endfunction
```

Write and source the file and try it out by visually selecting some text with a special character in it and pressing `<leader>g`. Vim will echo a version of the selected text suitable for passing to a shell command.

33.7 Running Grep

We're finally ready to add the `grep!` command that will perform the actual search. Replace the `echom` line so the code looks like this:

```
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>

function! GrepOperator(type)
    if a:type ==# 'v'
        normal! `<v`>y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    silent execute "grep! -R " . shellescape(@@) . " ."
    copen
endfunction
```

This should look familiar. We simply execute the `silent execute "grep! ..."` command we came up with in the last chapter. It's even more readable here because we're not trying to stuff the entire thing into a `nnoremap` command!

Write and source the file, then try it out and enjoy the fruits of your labor!

Because we've defined a brand new Vim operator we can use it in a lot of different ways, such as:

- `viw<leader>g`: Visually select a word, then grep for it.
- `<leader>g4w`: Grep for the next four words.
- `<leader>gt;`: Grep until semicolon.
- `<leader>gi[`: Grep inside square brackets.

This highlights one of the best things about Vim: its editing commands are like a language. When you add a new verb it automatically works with (most of) the existing nouns and adjectives.

33.8 Exercises

Read `:help visualmode()`.

Read `:help c_ctrl-u`.

Read `:help operatorfunc`.

Read `:help map-operator`.

34 Case Study: Grep Operator, Part Three

Our shiny new "grep operator" is working great, but part of writing Vimscript is being considerate and making your users' lives easier. We can do two more things to make our operator play nicely in the Vim ecosystem.

34.1 Saving Registers

By yanking the text into the unnamed register we destroy anything that was previously in there.

This isn't very nice to our users, so let's save the contents of that register before we yank and restore it after we've done. Change the code to look like this:

```
nnoremap <leader>g :set operatorfunc=GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>

function! GrepOperator(type)
    let saved_unnamed_register = @@

    if a:type ==# 'v'
        normal! `<v`>y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    silent execute "grep! -R " . shellescape(@@) . " ."
    copen

    let @@ = saved_unnamed_register
endfunction
```

We've added two `let` statements at the top and bottom of the function. The first saves the contents of `@@` into a variable and the second restores it.

Write and source the file. Make sure it works by yanking some text, then pressing `<leader>giw` to run our operator, then pressing `p` to paste the text you yanked before.

When writing Vim plugins you should *always* strive to save and restore any settings or registers your code modifies so you don't surprise and confuse your users.

34.2 Namespacing

Our script created a function named `GrepOperator` in the global namespace. This probably isn't a big deal, but when you're writing Vimscript it's far better to be safe than sorry.

We can avoid polluting the global namespace by tweaking a couple of lines in our code. Edit the file to look like this:

```
nnoremap <leader>g :set operatorfunc=<SID>GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call <SID>GrepOperator(visualmode())<cr>

function! s:GrepOperator(type)
    let saved_unnamed_register = @@

    if a:type ==# 'v'
        normal! `<v`>y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    silent execute "grep! -R " . shellescape(@@) . " ."
    copen

    let @@ = saved_unnamed_register
endfunction
```

The first three lines of the script have changed. First, we modified the function name to start with `s:` which places it in the current script's namespace.

We also modified the mappings and prepended the `GrepOperator` function name with `<SID>` so they could find the function. If we hadn't done this they would have tried to find the function in the global namespace, which wouldn't have worked.

Congratulations, our `grep-operator.vim` script is not only extremely useful, but it's also a considerate Vimscript citizen!

34.3 Exercises

Read `:help <SID>`.

Treat yourself to a snack. You deserve it!

35 Lists

We've worked a lot with variables so far, but we haven't talked about aggregates at all yet! Vim has two main aggregate types, and we'll look at the first now: lists.

Vimscript lists are ordered, heterogeneous collections of elements. Run the following command:

```
:echo ['foo', 3, 'bar']
```

Vim displays the list. Lists can of course be nested. Run the following command:

```
:echo ['foo', [3, 'bar']]
```

Vim happily displays the list.

35.1 Indexing

Vimscript lists are zero-indexed, and you can get at the elements in the usual way. Run this command:

```
:echo [0, [1, 2]][1]
```

Vim displays `[1, 2]`. You can also index from the end of the list, much like Python. Try this command:

```
:echo [0, [1, 2]][-2]
```

Vim displays `0`. The index `-1` refers to the last element in the list, `-2` is the second-to-last, and so on.

35.2 Slicing

Vim lists can also be sliced. This will *look* familiar to Python programmers, but it does *not* always act the same way! Run this command:

```
:echo ['a', 'b', 'c', 'd', 'e'][0:2]
```

Vim displays `['a', 'b', 'c']` (elements 0, 1 and 2). You can safely exceed the upper bound as well. Try this command:

```
:echo ['a', 'b', 'c', 'd', 'e'][0:100000]
```

Vim simply displays the entire list.

Slice indexes can be negative. Try this command:

```
:echo ['a', 'b', 'c', 'd', 'e'][-2:-1]
```

Vim displays `['d', 'e']` (elements -2 and -1).

When slicing lists you can leave off the first index to mean "the beginning" and/or the last index to mean "the end". Run the following commands:

```
:echo ['a', 'b', 'c', 'd', 'e'][:1]
:echo ['a', 'b', 'c', 'd', 'e'][3:]
```

Vim displays `['a', 'b']` and `['d', 'e']`.

Like Python, Vimscript allows you to index and slice strings too. Run the following command:

```
:echo "abcd"[0:2]
```

Vim displays `abc`. However, you can't use negative bare indices with strings. You *can* use negative indices when slicing strings though! Run the following command:

```
:echo "abcd"[-1] . "abcd"[-2:]
```

Vim displays `cd` (using a negative index silently resulted in an empty string).

35.3 Concatenation

You can combine Vim lists with `+`. Try this command:

```
:echo ['a', 'b'] + ['c']
```

Vim, unsurprisingly, displays `['a', 'b', 'c']`. There's not much else to say here – Vimscript lists are surprisingly sane compared to the rest of the language.

35.4 List Functions

Vim has a number of built-in functions for working with lists. Run these commands:

```
:let foo = ['a']
:call add(foo, 'b')
:echo foo
```

Vim mutates the list `foo` in-place to append `'b'` and displays `['a', 'b']`. Now run this command:

```
:echo len(foo)
```

Vim displays `2`, the length of the list. Try these commands:

```
:echo get(foo, 0, 'default')
:echo get(foo, 100, 'default')
```

Vim displays `a` and `default`. The `get` function will get the item at the given index from the given list, or return the given default value if the index is out of range in the list.

Run this command:

```
:echo index(foo, 'b')
:echo index(foo, 'nope')
```

Vim displays `1` and `-1`. The `index` function returns the first index of the given item in the given list, or `-1` if the item is not in the list.

Now run this command:

```
:echo join(foo)
:echo join(foo, '---')
:echo join([1, 2, 3], '')
```

Vim displays `a b`, `a---b`, and `123`. `join` will join the items in the given list together into a string, separated by the given separator string (or a space if none is given), coercing each item to a string if necessary/possible.

Run the following commands:

```
:call reverse(foo)
:echo foo
:call reverse(foo)
:echo foo
```

Vim displays `['b', 'a']` and then `['a', 'b']`. `reverse` reverses the given list *in place*.

35.5 Exercises

Read `:help List`. All of it. Notice the capital `L`.

Read `:help add()`.

Read `:help len()`.

Read `:help get()`.

Read `:help index()`.

Read `:help join()`.

Read `:help reverse()`.

Skim `:help functions` to find some other list-related functions I haven't mentioned yet. Run `:match Keyword /\clist/` to case-insensitively highlight the word `list` to make it easier to find what you're looking for.

36 Looping

You might be surprised to realize that we've gone through thirty five chapters of a programming language book without even mentioning loops! Vimscript offers so many other options for performing actions on text (`normal!`, etc) that loops aren't as necessary as they are in most other languages.

Even so, you'll definitely need them some day, so now we'll take a look at the two main kinds of loops Vim supports.

36.1 For Loops

The first kind of loop is the `for` loop. This may seem odd if you're used to Java, C or Javascript `for` loops, but turns out to be quite elegant. Run the following commands:

```
:let c = 0

:for i in [1, 2, 3, 4]
:  let c += i
:endfor

:echom c
```

Vim displays `10`, which is the result of adding together each element in the list. Vimscript `for` loops iterate over lists (or dictionaries, which we'll cover later).

There's no equivalent to the C-style `for (int i = 0; i < foo; i++)` loop form in Vimscript. This might seem bad at first, but in practice you'll never miss it.

36.2 While Loops

Vim also supports the classic `while` loop. Run the following commands:

```
:let c = 1
:let total = 0

:while c <= 4
:  let total += c
:  let c += 1
:endwhile

:echom total
```

Once again Vim displays `10`. This loop should be familiar to just about anyone who's programmed before, so we won't spend any time on it. You won't use it very often. Keep it in the back of your mind for the rare occasions that you want it.

36.3 Exercises

Read `:help for`.

Read `:help while`.

37 Dictionaries

The last type of Vimscript variable we'll talk about is the dictionary. Vimscript dictionaries are similar to Python's dicts, Ruby's hashes, and Javascript's objects.

Dictionaries are created using curly brackets. Values are heterogeneous, but *keys are always coerced to strings*. You didn't think things were going to be *completely* sane, did you?

Run this command:

```
:echo {'a': 1, 100: 'foo'}
```

Vim displays `{'a': 1, '100': 'foo'}`, which shows that Vimscript does indeed coerce keys to strings while leaving values alone.

Vimscript avoids the stupidity of the Javascript standard and lets you use a comma after the last element in a dictionary. Run the following command:

```
:echo {'a': 1, 100: 'foo',}
```

Once again Vim displays `{'a': 1, '100': 'foo'}`. You should *always* use the trailing comma in your dictionaries, *especially* when they're defined on multiple lines, because it will make adding new entries less error-prone.

37.1 Indexing

To look up a key in a dictionary you use the same syntax as most languages. Run this command:

```
:echo {'a': 1, 100: 'foo',}['a']
```

Vim displays `1`. Try it with a non-string index:

```
:echo {'a': 1, 100: 'foo',}[100]
```

Vim coerces the index to a string before performing the lookup, which makes sense since keys can only ever be strings.

Vimscript also supports the Javascript-style "dot" lookup when the key is a string consisting only of letters, digits and/or underscores. Try the following commands:

```
:echo {'a': 1, 100: 'foo',}.a
:echo {'a': 1, 100: 'foo',}.100
```

Vim displays the correct element in both cases. How you choose to index your dictionaries is a matter of taste and style.

37.2 Assigning and Adding

Adding entries to dictionaries is done by simply assigning them like variables. Run this command:

```
:let foo = {'a': 1}
:let foo.a = 100
:let foo.b = 200
:echo foo
```

Vim displays `{'a': 100, 'b': 200}`, which shows that assigning and adding entries both work the same way.

37.3 Removing Entries

There are two ways to remove entries from a dictionary. Run the following commands:

```
:let test = remove(foo, 'a')
:unlet foo.b
:echo foo
:echo test
```

Vim displays `{}` and `100`. The `remove` function will remove the entry with the given key from the given dictionary and return the removed value. The `unlet` command also removes dictionary entries, but you can't use the value.

You cannot remove nonexistent entries from a dictionary. Try running this command:

```
:unlet foo["asdf"]
```

Vim throws an error.

The choice of `remove` or `unlet` is mostly a matter of personal taste. If pressed I'd recommend using `remove` everywhere because it's more flexible than `unlet`. `remove` can do anything `unlet` can do but the reverse isn't true, so you can always be consistent if you use `remove`.

37.4 Dictionary Functions

Like lists, Vim has a number of built-in functions for working with dictionaries. Run the following command:

```
:echom get({'a': 100}, 'a', 'default')
:echom get({'a': 100}, 'b', 'default')
```

Vim displays `100` and `default`, just like the `get` function for lists.

You can also check if a given key is present in a given dictionary. Run this commands:

```
:echom has_key({'a': 100}, 'a')
:echom has_key({'a': 100}, 'b')
```

Vim displays `1` and `0`. Remember that Vimscript treats `0` as falsy and any other number as truthy.

You can pull the key-value pairs out of a dictionary with `items`. Run this command:

```
:echo items({'a': 100, 'b': 200})
```

Vim will display a nested list that looks something like `[['a', 100], ['b', 200]]`. As far as I can tell Vimscript dictionaries are *not* guaranteed to be ordered, so don't expect that the items you get out of an `items` call will be in a specific order!

You can get just the keys or just the values with the `keys` and `values` functions. They work as expected – look them up.

37.5 Exercises

Read `:help Dictionary`. All of it. Notice the capital `D`.

Read `:help get()`.

Read `:help has_key()`.

Read `:help items()`.

Read `:help keys()`.

Read `:help values()`.

38 Toggling

In one of the first chapters we talked about how to set options in Vim. For boolean options we can use `set someoption!` to "toggle" the option. This is especially nice when we create a mapping for that command.

Run the following command:

```
:nnoremap <leader>N :setlocal number!<cr>
```

Try it out by pressing `<leader>N` in normal mode. Vim will toggle the line numbers for the current window off and on. Creating a "toggle" mapping like this is really handy, because we don't need to have two separate keys to turn something off and on.

Unfortunately this only works for boolean options. If we want to toggle a non-boolean option we'll need to do a bit more work.

38.1 Toggling Options

Let's start by creating a function that will toggle an option for us, and a mapping that will call it. Put the following into your `~/.vimrc` file (or a separate file in `~/.vim/plugin/` if you prefer):

```
nnoremap <leader>f :call FoldColumnToggle()<cr>

function! FoldColumnToggle()
    echom &foldcolumn
endfunction
```

Write and source the file, then try it out by pressing `<leader>f` Vim will display the current value of the `foldcolumn` option. Go ahead and read `:help foldcolumn` if you're unfamiliar with this option.

Let's add in the actual toggling functionality. Edit the code to look like this:

```
nnoremap <leader>f :call FoldColumnToggle()<cr>

function! FoldColumnToggle()
    if &foldcolumn
        setlocal foldcolumn=0
    else
        setlocal foldcolumn=4
    endif
endfunction
```

Write and source the file and try it out. Each time you press it Vim will either show or hide the fold column.

The `if` statement simply checks if `&foldcolumn` is truthy (remember that Vim treats the integer `0` as falsy and any other number as truthy). If so, it sets it to zero (which hides it). Otherwise it sets it to four. Pretty simple.

You can use a simple function like this to toggle any option where `0` means "off" and any other number is "on".

38.2 Toggling Other Things

Options aren't the only thing we might want to toggle. One particularly nice thing to have a mapping for is the quickfix window. Let's start with the same skeleton as before. Add the following code to your file:

```
nnoremap <leader>q :call QuickfixToggle()<cr>

function! QuickfixToggle()
    return
endfunction
```

This mapping doesn't do anything yet. Let's transform it into something slightly more useful (but not completely finished yet). Change the code to look like this:

```
nnoremap <leader>q :call QuickfixToggle()<cr>

function! QuickfixToggle()
    copen
endfunction
```

Write and source the file. If you try out the mapping now you'll see that it simply opens the quickfix window.

To get the "toggling" behavior we're looking for we'll use a quick, dirty solution: a global variable. Change the code to look like this:

```
nnoremap <leader>q :call QuickfixToggle()<cr>

function! QuickfixToggle()
    if g:quickfix_is_open
        cclose
        let g:quickfix_is_open = 0
    else
        copen
        let g:quickfix_is_open = 1
    endif
endfunction
```

What we've done is pretty simple – we're simply storing a global variable describing the open/closed state of the quickfix window whenever we call the function.

Write and source the file, and try to run the mapping. Vim will complain that the variable is not defined yet! Let's fix that by initializing it once:

```
nnoremap <leader>q :call QuickfixToggle()<cr>

let g:quickfix_is_open = 0

function! QuickfixToggle()
    if g:quickfix_is_open
        cclose
        let g:quickfix_is_open = 0
    else
        copen
        let g:quickfix_is_open = 1
    endif
endfunction
```

Write and source the file, and try the mapping. It works!

38.3 Improvements

Our toggle function works, but has a few problems.

The first is that if the user manually opens or closes the window with `:copen` or `:cclose` our global variable doesn't get updated. This isn't really a huge problem in practice because most of the time the user will probably be opening the window with the mapping, and if not they can always just press it again.

This illustrates an important point about writing Vimscript code: if you try to handle every single edge case you'll get bogged down in it and never get any work done.

Getting something that works most of the time (and doesn't explode when it doesn't work) and getting back to coding is usually better than spending hours getting it 100% perfect. The exception is when you're writing a plugin you expect many people to use. In that case it's best to spend the time and make it bulletproof to keep your users happy and reduce bug reports.

38.4 Restoring Windows/Buffers

The other problem with our function is that if the user runs the mapping when they're already in the quickfix window, Vim closes it and dumps them into the last split instead of sending them back where they were. This is annoying if you just want to check the quickfix window really quick and get back to working.

To solve this we'll introduce an idiom that comes in handy a lot when writing Vim plugins. Edit your code to look like this:

```
nnoremap <leader>q :call QuickfixToggle()<cr>

let g:quickfix_is_open = 0

function! QuickfixToggle()
    if g:quickfix_is_open
        cclose
        let g:quickfix_is_open = 0
        execute g:quickfix_return_to_window . "wincmd w"
    else
        let g:quickfix_return_to_window = winnr()
        copen
        let g:quickfix_is_open = 1
    endif
endfunction
```

We've added two new lines in this mapping. One of them (in the `else` clause) sets another global variable which saves the current window number before we run `:copen`.

The second line (in the `if` clause) executes `wincmd w` with that number prepended as a count, which tells Vim to go to that window.

Once again our solution isn't bulletproof, because the user might open or close new split between runs of the mapping. Even so, it handles the majority of cases so it's good enough for now.

This strategy of manually saving global state would be frowned upon in most serious programs, but for tiny little Vimscript functions it's a quick and dirty way of getting something mostly working and moving on with your life.

38.5 Exercises

Read `:help foldcolumn`.

Read `:help winnr()`

Read `:help ctrl-w_w`.

Read `:help wincmd`.

Namespace the functions by adding `s:` and `<SID>` where necessary.

39 Functional Programming

We're going to take a short break now to talk about a style of programming you may have heard of: functional programming[1].

If you've programmed in languages like Python, Ruby or Javascript, or *especially* Lisp, Scheme, Clojure or Haskell, you're probably familiar with the idea of using functions as variables and using data structures with immutable state. If you've never done this before you can safely skip this chapter, but I'd encourage you to try it anyway and broaden your horizons!

Vimscript has all the pieces necessary to program in a kind-of-functional style, but it's a bit clunky. We can create a few helpers that will make it less painful though.

Go ahead and create a `functional.vim` file somewhere so you don't have to keep typing everything over and over. This file will be our scratchpad for this chapter.

39.1 Immutable Data Structures

Unfortunately Vim doesn't have any immutable collections like Clojure's vectors and maps built-in, but by creating some helper functions we can fake it to some degree.

Add the following function to your file:

```
function! Sorted(l)
    let new_list = deepcopy(a:l)
    call sort(new_list)
    return new_list
endfunction
```

Source and write the file, then run `:echo Sorted([3, 2, 4, 1])` to try it out. Vim echoes `[1, 2, 3, 4]`.

How is this different from simply calling the built-in `sort()` function? The key is the first line: `let new_list = deepcopy(a:l)`. Vim's `sort()` sorts the list *in place*, so we first create a full copy of the list and sort *that* so the original list won't be changed.

This prevents side effects and helps us write code that is easier to reason about and test. Let's add a few more helper functions in this same style:

[1] https://secure.wikimedia.org/wikipedia/en/wiki/Functional_programming

```
function! Reversed(l)
    let new_list = deepcopy(a:l)
    call reverse(new_list)
    return new_list
endfunction

function! Append(l, val)
    let new_list = deepcopy(a:l)
    call add(new_list, a:val)
    return new_list
endfunction

function! Assoc(l, i, val)
    let new_list = deepcopy(a:l)
    let new_list[a:i] = a:val
    return new_list
endfunction

function! Pop(l, i)
    let new_list = deepcopy(a:l)
    call remove(new_list, a:i)
    return new_list
endfunction
```

Each of these functions is exactly the same except for the middle line and the arguments they take. Source and write the file and try them out on a few lists.

`Reversed()` takes a list and returns a new list with the elements reversed.

`Append()` returns a new list with the given value appended to the end of the old one.

`Assoc()` (short for "associate") returns a new list, with the element at the given index replaced by the new value.

`Pop()` returns a new list with the element at the given index removed.

39.2 Functions as Variables

Vimscript supports using variables to store functions, but the syntax is a bit obtuse. Run the following commands:

```
:let Myfunc = function("Append")
:echo Myfunc([1, 2], 3)
```

Vim will display `[1, 2, 3]` as expected. Notice that the variable we used started with a capital letter. If a Vimscript variable refers to a function it must start with a capital letter.

Functions can be stored in lists just like any other kind of variable. Run the following commands:

```
:let funcs = [function("Append"), function("Pop")]
:echo funcs[1](['a', 'b', 'c'], 1)
```

Vim displays `['a', 'c']`. The `funcs` variable does *not* need to start with a capital letter because it's storing a list, not a function. The contents of the list don't matter at all.

39.3 Higher-Order Functions

Let's create a few of the most commonly-used higher-order functions. If you're not familiar with that term, higher-order functions are functions that take *other* functions and do something with them.

We'll begin with the trusty `map` function. Add this to your file:

```
function! Mapped(fn, l)
    let new_list = deepcopy(a:l)
    call map(new_list, string(a:fn) . '(v:val)')
    return new_list
endfunction
```

Source and write the file, and try it out by running the following commands:

```
:let mylist = [[1, 2], [3, 4]]
:echo Mapped(function("Reversed"), mylist)
```

Vim displays `[[2, 1], [4, 3]]`, which is the result of applying `Reversed()` to every element in the list.

How does `Mapped()` work? Once again we create a fresh list with `deepcopy()`, do something to it, and return the modified copy – nothing new there. The tricky part is the middle.

`Mapped()` takes two arguments: a funcref (Vim's term for "variable holding a function") and a list. We use the built-in `map()` function to perform the actual work. Read `:help map()` now to see how it works.

Now we'll create a few other common higher-order functions. Add the following to your file:

```
function! Filtered(fn, l)
    let new_list = deepcopy(a:l)
    call filter(new_list, string(a:fn) . '(v:val)')
    return new_list
endfunction
```

Try `Filtered()` out with the following commands:

```
:let mylist = [[1, 2], [], ['foo'], []]
:echo Filtered(function('len'), mylist)
```

Vim displays `[[1, 2], ['foo']]`.

`Filtered()` takes a predicate function and a list. It returns a copy of the list that contains only the elements of the original where the result of calling the function on it is "truthy". In this case we use the built-in `len()` function, so it filters out all the elements whose length is zero.

Finally we'll create the counterpart to `Filtered()`:

```
function! Removed(fn, l)
    let new_list = deepcopy(a:l)
    call filter(new_list, '!' . string(a:fn) . '(v:val)')
    return new_list
endfunction
```

Try it out just like we did with `Filtered()`:

```
:let mylist = [[1, 2], [], ['foo'], []]
:echo Removed(function('len'), mylist)
```

Vim displays `[[], []]`. `Removed()` is like `Filtered()` except it only keeps elements where the predicate function does *not* return something truthy.

The only difference in the code is the single `'!' .` we added to the call command, which negates the result of the predicate.

39.4 Performance

You might be thinking that copying lists all over the place is wasteful, since Vim has to constantly create new copies and garbage collect old ones.

If so: you're right! Vim's lists don't support the same kind of structural sharing as Clojure's vectors, so all those copy operations can be expensive.

Sometimes this will matter. If you're working with enormous lists, things can slow down. In real life, though, you might be surprised at how little you'll actually notice the difference.

Consider this: as I'm writing this chapter my Vim program is using about 80 megabytes of memory (and I have a *lot* of plugins installed). My laptop has 8 *gigabytes* of memory in it. Is the overhead of having a few copies of a list around really going to make a noticeable difference? Of course that depends on the size of the list, but in most cases the answer will be "no".

To contrast, my Firefox instance with five tabs open is currently using 1.22 *gigabytes* of RAM.

You'll need to use your own judgement about when this style of programming creates unacceptably bad performance.

39.5 Exercises

Read `:help sort()`.

Read `:help reverse()`.

Read `:help copy()`.

Read `:help deepcopy()`.

Read `:help map()` if you haven't already.

Read `:help function()`.

Modify `Assoc()`, `Pop()`, `Mapped()`, `Filtered()` and `Removed()` to support dictionaries. You'll probably need `:help type()` for this.

Implement `Reduced()`.

Pour yourself a glass of your favorite drink. This chapter was intense!

40 Paths

Vim is a text editor, and text editors (usually) work with text files. Text files live on filesystems, and to specify files we use paths. Vimscript has a few built-in utilities that can be extremely helpful when you need to work with paths.

40.1 Absolute Paths

Sometimes it's handy to be able to get the absolute path of a certain file for use with external scripts. Run the following commands:

```
:echom expand('%')
:echom expand('%:p')
:echom fnamemodify('foo.txt', ':p')
```

The first command displays the relative path of whatever file you're currently editing. `%` means "the current file". Vim supports a bunch of other strings you can use with `expand()` as well.

The second command displays the full, absolute path of that file. The `:p` in the string tells Vim that you want the absolute path. There are a ton of other modifiers you can use.

The third command displays an absolute path to the file `foo.txt` in the current directory, regardless of whether that file actually exists. `fnamemodify()` is a Vim function that's more flexible than `expand()` in that you can specify any file name, not just one of `expand()`'s special strings.

40.2 Listing Files

You might also want to get a listing of files in a specific directory. Run the following command:

```
:echo globpath('.', '*')
```

Vim will display all of the files and directories in the current directory. The `globpath()` function returns a string, with each name separated by a newline. To get a list you'll need to `split()` it yourself. Run this command:

```
:echo split(globpath('.', '*'), '\n')
```

This time Vim displays a Vimscript list containing each path. If you've got newlines in your filenames you're on your own, sorry.

`globpath()`'s wildcards work mostly as you would expect. Run the following command:

```
:echo split(globpath('.', '*.txt'), '\n')
```

Vim displays a list of all `.txt` files in the current directory.

You can recursively list files with `**`. Run this command:

```
:echo split(globpath('.', '**'), '\n')
```

Vim will list all files and directories under the current directory.

`globpath()` is *extremely* powerful. You'll learn more when you complete this chapter's exercises.

40.3 Exercises

Read `:help expand()`.

Read `:help fnamemodify()`.

Read `:help filename-modifiers`.

Read `:help simplify()`.

Read `:help resolve()`.

Read `:help globpath()`.

Read `:help wildcards`.

41 Creating a Full Plugin

We've covered a lot of ground in the last forty or so chapters. In the final part of the book we're going to walk through creating a Vim plugin for a programming language from scratch.

This is not for the faint of heart. It's going to take a lot of effort.

If you want to stop now, that's completely fine! You've already learned enough to make serious enhancements to your own `~/.vimrc` file and to fix bugs you find in other people's plugins.

There's no shame in saying "that's enough for me, I don't want to spend hours of my life creating plugins I won't use very often". Be practical. If you can't think of a full plugin you want to make, stop now and come back when you do.

If you *do* want to continue make sure you're ready to devote some time. The rest of the book is going to be intense, and I'm going to assume you actually want to *learn*, not just coast through the chapters reading them on your couch.

41.1 Potion

The plugin we're going to make is going to add support for the Potion[1] programming language.

Potion is a toy language created by _why the lucky stiff before his disappearance. It's an extremely small language which makes it ideal for our purposes.

Potion feels a lot like Io[2], with some ideas from Ruby, Lua, and other languages mixed in. If you've never tried Io it may seem weird. I strongly recommend playing with Potion for at least an hour or two. You won't use it in real life, but it might change the way you think and expose you to new ideas.

The current implementation of Potion has a lot of rough edges. For example: if you mess up the syntax it usually segfaults. Try not to get too hung up on this. I'll provide you with lots of sample code that works so you can focus mostly on the Vimscript and not Potion.

The goal is not to learn Potion (though that can be fun too). The goal is to use Potion as a small example so we can touch on a lot of different aspects of writing full Vim plugins.

41.2 Exercises

Download and install Potion[3]. You're on your own for this one. It should be fairly simple.

[1] http://fogus.github.com/potion/index.html
[2] http://iolanguage.com/
[3] http://fogus.github.com/potion/index.html

Make sure you can get the first couple examples in the pamphlet working in the Potion interpreter and by putting them in a `.pn` file. If it seems like the interpreter isn't working check out this issue[4] for a possible cause.

[4] https://github.com/fogus/potion/issues/12

42 Plugin Layout in the Dark Ages

The first thing we need to talk about is how to structure our plugin. In the past this was a messy affair, but now there's a tool that makes the process of installing Vim plugins much, *much* saner.

We need to go over the basic layout first, then we'll talk about how to restore our sanity.

42.1 Basic Layout

Vanilla Vim supports splitting plugins into multiple files. There are a number of different directories you can create under `~/.vim` that mean various things.

We'll cover the most important directories now, but don't stress out too much about them. We'll go over them one at a time as we create our Potion plugin.

Before we continue we need to talk about some vocabulary.

I've been using the word "plugin" to mean "a big ol' hunk of Vimscript that does a bunch of related stuff". Vim has a more specific meaning of "plugin", which is "a file in `~/.vim/plugins/`".

Most of the time I'll be using the first definition. I'll try to be clear when I mean the second.

42.2 ~/.vim/colors/

Files inside `~/.vim/colors/` are treated as color schemes. For example: if you run `:color mycolors` Vim will look for a file at `~/.vim/colors/mycolors.vim` and run it. That file should contain all the Vimscript commands necessary to generate your color scheme.

We're not going to cover color schemes in this book. If you want to create your own you should copy an existing scheme and modify it. Remember that `:help` is your friend.

42.3 ~/.vim/plugin/

Files inside `~/.vim/plugin/` will each be run once *every time* Vim starts. These files are meant to contain code that you always want loaded whenever you start Vim.

42.4 ~/.vim/ftdetect/

Any files in `~/.vim/ftdetect/` will *also* be run every time you start Vim.

`ftdetect` stands for "filetype detection". The files in this directory should set up autocommands that detect and set the `filetype` of files, and *nothing else*. This means they should never be more than one or two lines long.

42.5 ~/.vim/ftplugin/

Files in `~/.vim/ftplugin/` are different.

The naming of these files matters! When Vim sets a buffer's `filetype` to a value it then looks for a file in `~/.vim/ftplugin/` that matches. For example: if you run `set filetype=derp` Vim will look for `~/.vim/ftplugin/derp.vim`. If that file exists, it will run it.

Vim also supports directories inside `~/.vim/ftplugin/`. To continue our example: `set filetype=derp` will also make Vim run any and all `*.vim` files inside `~/.vim/ftplugin/derp/`. This lets you split up your plugin's `ftplugin` files into logical groups.

Because these files are run every time a buffer's `filetype` is set they *must* only set buffer-local options! If they set options globally they would overwrite them for all open buffers!

42.6 ~/.vim/indent/

Files in `~/.vim/indent/` are a lot like `ftplugin` files. They get loaded based on their names.

`indent` files should set options related to indentation for their filetypes, and those options should be buffer-local.

Yes, you could simply put this code in the `ftplugin` files, but it's better to separate it out so other Vim users will understand what you're doing. It's just a convention, but please be a considerate plugin author and follow it.

42.7 ~/.vim/compiler/

Files in `~/.vim/compiler/` work exactly like `indent` files. They should set compiler-related options in the current buffer based on their names.

Don't worry about what "compiler-related options" means right now. We'll cover that later.

42.8 ~/.vim/after/

The `~/.vim/after/` directory is a bit of a hack. Files in this directory will be loaded every time Vim starts, but *after* the files in `~/.vim/plugin/`.

This allows you to override Vim's internal files. In practice you'll rarely need this, so don't worry about it until you find yourself thinking "Vim itself sets option `x`, but I want something different".

42.9 ~/.vim/autoload/

The `~/.vim/autoload/` directory is an incredibly important hack. It sounds a lot more complicated than it actually is.

In a nutshell: `autoload` is a way to delay the loading of your plugin's code until it's actually needed. We'll cover this in more detail later when we refactor our plugin's code to take advantage of it.

42.10 ~/.vim/doc/

Finally, the `~/.vim/doc/` directory is where you can add documentation for your plugin. Vim has a huge focus on documentation (as evidenced by all the `:help` commands we've been running) so it's important to document your plugins.

42.11 Exercises

Reread this chapter. I'm not kidding. Make sure you understand (in a very rough way) what each directory we've talked about does.

For extra credit, find some Vim plugins you use and look at how they structure their files.

43 A New Hope: Plugin Layout with Pathogen

Vim's vanilla layout for plugin files makes sense if you're just adding a file here and there to customize your own Vim experience, but turns into a mess when you want to use plugins other people have written.

In the past, when you wanted to use a plugin someone else wrote you would download the files and place them, one-by-one, into the appropriate directories. You could also use `zip` or `tar` to do the placing for you.

There are a few significant problems with this approach:

- What happens when you want to update a plugin? You can overwrite the old files, but how do you know if the author deleted a file that you now need to delete by hand?
- What if two plugins happen to have a file with the same name (like `utils.vim` or something generic like that)? Sometimes you can simply rename it, but if it's in `autoload/` or another directory where the names matter you've got to edit the plugin yourself. Not fun.

People came up with several hacks to try to make things easier, like Vimballs. Luckily we don't need to suffer through these ugly hacks any more. Tim Pope[1] created the wonderful Pathogen[2] plugin that makes managing multiple plugins a breeze, as long as plugin authors structure their plugins in a sane way.

Let's take a quick look at how Pathogen works and what we need to do to make our plugin compatible.

43.1 Runtimepath

When Vim looks for files in a specific directory, like `syntax/`, it doesn't just look in a single place. Much like `PATH` on Linux/Unix/BSD systems, Vim has the `runtimepath` setting which tells it where to find files to load.

Create a `colors` directory on your Desktop. Create a file in that directory called `mycolor.vim` (you can leave it empty for this demonstration). Open Vim and run this command:

```
:color mycolor
```

Vim will display an error, because it doesn't know to look on your Desktop for files. Now run this command:

[1] http://tpo.pe/

[2] http://www.vim.org/scripts/script.php?script_id=2332

```
:set runtimepath=/Users/sjl/Desktop
```

You'll need to change the path to match the path of your own Desktop, of course. Now try the color command again:

```
:color mycolor
```

This time Vim doesn't throw an error, because it was able to find the `mycolor.vim` file. Because the file was blank it didn't actually *do* anything, but we know it was found because it didn't throw an error.

43.2 Pathogen

The Pathogen plugin automatically adds paths to your `runtimepath` when you load Vim. Any directories inside `~/.vim/bundle/` will each be added to the `runtimepath`.

This means that each directory inside `bundle/` should contain some or all of the standard Vim plugin directories, like `colors/` and `syntax/`. Vim can now load files from each of those directories, which makes it simple to keep each plugin's files in its own directory.

This makes it trivial to update plugins. You can simply blow away the old plugin's directory entirely and replace it with the new version. If you keep your `~/.vim` directory under version control (and you should) you can use Mercurial's subrepos or Git's submodules to directly check out each plugin's repository into `~/.vim/bundle/` and then update it with a simple `hg pull; hg update` or `git pull origin master`.

43.3 Being Pathogen-Compatible

When we write our Potion plugin we want to let our users use it with Pathogen. Doing this is simple: we simply put our files in the appropriate directories inside the plugin's repository!

Our plugin's repository will wind up looking like this:

```
potion/
    README
    LICENSE
    doc/
        potion.txt
    ftdetect/
        potion.vim
    ftplugin/
        potion.vim
    syntax/
        potion.vim
    ... etc ...
```

We can put this on GitHub or Bitbucket and users can simply clone it down into `bundle/` and everything will just work!

43.4 Exercises

Install Pathogen[3] if you haven't already done so.

Create a Mercurial or Git repository for your plugin, called `potion`. You can put it anywhere you like and symlink it into `~/.vim/bundle/potion/` or just put it directly in `~/.vim/bundle/potion/`.

Create `README` and `LICENSE` files in the repository and commit them.

Push the repository up to Bitbucket or GitHub.

Read `:help runtimepath`.

[3] http://www.vim.org/scripts/script.php?script_id=2332

44 Detecting Filetypes

Let's create a Potion file we can use as a sample as we're working on our plugin. Create a `factorial.pn` file somewhere and put the following Potion code inside it:

```
factorial = (n):
    total = 1
    n to 1 (i):
        total *= i.
    total.

10 times (i):
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.
```

This code creates a simple factorial function and calls it ten times, printing the results each time. Go ahead and run it with `potion factorial.pn`. The output should look like this:

```
0! is: 0
1! is: 1
2! is: 2
3! is: 6
4! is: 24
5! is: 120
6! is: 720
7! is: 5040
8! is: 40320
9! is: 362880
```

If you don't get this output, or you get an error, stop and figure out what's gone wrong. The code should work exactly as-is.

Take some time to understand how the code works. Refer to the Potion docs liberally. It's not critical to understanding Vimscript but it will make you a better programmer.

44.1 Detecting Potion Files

Open `factorial.pn` in Vim and run the following command:

```
:set filetype?
```

Vim will display `filetype=` because it doesn't know what a `.pn` file is yet. Let's fix that.

Create `ftdetect/potion.vim` in your plugin's repo. Put the following lines into it:

```
au BufNewFile,BufRead *.pn set filetype=potion
```

This creates a single autocommand: a command to set the filetype of `.pn` files to `potion`. Pretty straightforward.

Notice that we *didn't* use an autocommand group like we usually would. Vim automatically wraps the contents of `ftdetect/*.vim` files in autocommand groups for you, so you don't need to worry about it.

Close the `factorial.pn` file and reopen it. Now run the previous command again:

```
:set filetype?
```

This time Vim displays `filetype=potion`. When Vim started up it loaded the autocommand group inside `~/.vim/bundle/potion/ftdetect/potion.vim`, and when it opened `factorial.pn` the autocommand fired, setting the `filetype` to `potion`.

Now that we've taught Vim to recognize Potion files we can move on to actually creating some useful behavior in our plugin.

44.2 Exercises

Read `:help ft`. Don't worry if you don't understand everything there.

Read `:help setfiletype`.

Modify the Potion plugin's `ftdetect/potion.vim` script to use `setfiletype` instead of `set filetype`.

45 Basic Syntax Highlighting

Now that we've gotten the boilerplate out of the way it's time to start writing some useful code for our Potion plugin. We'll start with some simple syntax highlighting.

Create a `syntax/potion.vim` file in your plugin's repo. Put the following code into the file:

```
if exists("b:current_syntax")
    finish
endif

echom "Our syntax highlighting code will go here."

let b:current_syntax = "potion"
```

Close Vim, and then open your `factorial.pn` file. You may or may not see the message, depending on whether you have any other plugins that perform commands after this one gets run. If you run `:messages` you'll definitely see that the file was indeed loaded.

Note: Whenever I tell you to open the Potion file I want you to do it in a *new Vim window/instance* instead of in a split/tab. Opening a new Vim window causes Vim to reload all your bundled files for that window, whereas using a split does not.

The lines at the beginning and end of the file are a convention that prevents it from being loaded if syntax highlighting has already been enabled for this buffer.

45.1 Highlighting Keywords

For the rest of this chapter we'll ignore the `if` and `let` boilerplate at the beginning and end of the file. Don't remove those lines, just forget about them.

Replace the placeholder `echom` in the file with the following code:

```
syntax keyword potionKeyword to times
highlight link potionKeyword Keyword
```

Close the `factorial.pn` file and reopen it. The `to` and `times` words will be highlighted as keywords in your color scheme!

These two lines show the basic structure of simple syntax highlighting in Vim. To highlight a piece of syntax:

- You first define a "chunk" of syntax using `syntax keyword` or a related command (which we'll talk about later).
- You then link "chunks" to highlighting groups. A highlighting group is something you define in a color scheme, for example "function names should be blue".

This lets plugin authors define the "chunks" of syntax in ways that make sense to them, and then link them to common highlighting groups. It also lets color scheme creators define colors for a common set of programming constructs so they don't need to know about individual languages.

Potion has a bunch of other keywords that we haven't used in our toy program, so let's edit our syntax file to highlight those too:

```
syntax keyword potionKeyword loop times to while
syntax keyword potionKeyword if elsif else
syntax keyword potionKeyword class return

highlight link potionKeyword Keyword
```

First of all: the last line hasn't changed. We're still telling Vim that anything in the `potionKeyword` syntax group should be highlighted as a `Keyword`.

We've now got three lines, each starting with `syntax keyword potionKeyword`. This shows that running this command multiple times doesn't *reset* the syntax group – it adds to it! This lets you define groups piecemeal.

How you define your groups is up to you:

- You might just toss everything onto one line and be done with it.
- You might prefer to break the lines up so they fit within 80 columns to make them easier to read.
- You could have a separate line for each item in a group, to make diffs looks nicer.
- You could do what I've done here and group related items together.

45.2 Highlighting Functions

Another standard Vim highlighting group is `Function`. Let's add some of the built-in Potion functions to our highlighting script. Edit the guts of your syntax file so it looks like this:

```
syntax keyword potionKeyword loop times to while
syntax keyword potionKeyword if elsif else
syntax keyword potionKeyword class return

syntax keyword potionFunction print join string

highlight link potionKeyword Keyword
highlight link potionFunction Function
```

Close and reopen `factorial.pn` and you'll see that the built-in potion functions are now highlighted.

This works exactly the same way as keyword highlighting. We've defined a new syntax group and linked it to a different highlighting group.

45.3 Exercises

Think about why the `if exists` and `let` lines at the beginning and end of the file are useful. If you can't figure it out, don't worry about it. I had to ask Tim Pope to be sure.

Skim over `:help syn-keyword`. Pay close attention to the part that mentions `iskeyword`.

Read `:help iskeyword`.

Read `:help group-name` to get an idea of some common highlighting groups that color scheme authors frequently use.

46 Advanced Syntax Highlighting

So far we've defined some simple syntax highlighting for Potion files: keywords and functions.

If you didn't do the exercises in the last chapter, you need to go back and do them. I'm going to assume you did them.

In fact, you should go back and do *any* exercises you skipped. Even if you think you don't need them, you *need* to do them for this book to be effective. Please trust me on this.

46.1 Highlighting Comments

One obvious part of Potion that we need to highlight is comments. The problem is that Potion comments start with `#` which is (almost always) not in `iskeyword`.

If you don't know what `iskeyword` means, you didn't listen to me. Go back and *do the damn exercises*. I'm not just throwing useless busywork at you when I write the exercises for each chapter. You *really* need to do them to understand the book.

Because `#` isn't a keyword character we need to use a regular expression to match it (and the rest of the comment). We'll do this with `syntax match` instead of `syntax keyword`. Add the following lines to your syntax file:

```
syntax match potionComment "\v#.*$"
highlight link potionComment Comment
```

I'm not going to tell you where to put them in the file any more. You're a programmer: use your judgement.

Close and reopen `factorial.pn`. Add a comment somewhere in the file and you'll see that it's highlighted as a comment.

The second line is simple: it tells Vim to highlight anything in the `potionComment` syntax group as a `Comment`.

The first line is something new. We use `syntax match` which tells Vim to match *regexes* instead of literal keywords.

Notice that the regular expression we're using starts with `\v` which tells Vim to use "very magic" mode. Reread the chapter on basic regular expressions if you're not sure what that means.

In this particular case the "very magic" mode isn't necessary. But in the future we might change this regex and wonder why it's not working, so I'd recommend *always* using "very magic" regexes for consistency.

As for the regex itself, it's fairly simple: comments start with a hash and include all characters from there to the end of the line.

If you need a refresher course on regular expressions you should take a look at Learn Regex the Hard Way[1] by Zed Shaw.

46.2 Highlighting Operators

Another part of Potion we need regexes to highlight is operators. Add the following to your syntax file:

```
syntax match potionOperator "\v\*"
syntax match potionOperator "\v/"
syntax match potionOperator "\v\+"
syntax match potionOperator "\v-"
syntax match potionOperator "\v\?"
syntax match potionOperator "\v\*\="
syntax match potionOperator "\v/\="
syntax match potionOperator "\v\+\="
syntax match potionOperator "\v-\="

highlight link potionOperator Operator
```

Close and reopen `factorial.pn`. Notice that the `*=` in the factorial function is now highlighted.

The first thing you probably noticed about this hunk of code is that I put each regex on its own line instead of grouping them like I did with keywords. This is because `syntax match` does *not* support multiple groups on a single line.

You should also note that I used `\v` at the beginning of every regular expression, even when it wasn't strictly necessary. I prefer to keep my regex syntax consistent when writing Vimscript, even if it means a few extra characters.

You might be wondering why I didn't use a regex like `"\v-\=?"` to match both `-` *and* `-=` in one line. You can absolutely do that if you want to. It will work just fine. I just tend to think of `-` and `-=` as separate operators, so I put them on separate lines.

Defining those operators as separate matches simplifies the regexes at the cost of some extra verbosity. I prefer doing it like that, but you may feel differently. Use your judgement.

I also never defined `=` as an operator. We'll do that in a second, but I wanted to avoid it for now so I could teach you a lesson.

Because I used separate regexes for `-` and `-=` I had to define `-=` *after* `-`!

If I did it in the opposite order and used `-=` in a Potion file, Vim would match `-` (and highlight it, of course) and only `=` would remain for matching. This shows when you're building groups with `syntax match` each group "consumes" pieces of the file that can't be matched later.

[1] http://regex.learncodethehardway.org/

This is an oversimplification, but I don't want to get bogged down in the details just yet. For now, your rule of thumb should be to match larger groups after smaller groups later because groups defined *later* have priority over groups defined *earlier*.

Let's go ahead and add = as an operator, now that we've had our lesson:

```
syntax match potionOperator "\v\="
```

Take a second and think about where you need to put this in the syntax file. Reread the last few paragraphs if you need a hint.

46.3 Exercises

Read `:help syn-match`.

Read `:help syn-priority`.

We didn't make `:` an operator in our example. Read the Potion docs and make a conscious decision about whether to make `:` an operator. If you decide to do so, add it to the syntax file.

Do the same for `.` and `/`.

Add a syntax group `potionNumber` that highlights numbers. Link it to the highlight group `Number`. Remember that Potion supports numbers like `2`, `0xffaf`, `123.23`, `1e-2`, and `1.9956e+2`. Remember to balance the time it takes to handle edge cases with the amount of time those edge cases will actually be used.

47 Even More Advanced Syntax Highlighting

Syntax highlighting in Vim is a topic that could easily fill a book of its own.

We're going to cover one last important part of it here and then move on to other things. If you want to learn more you can read the Vim documentation with `:help syntax` and look at syntax files other people have made.

47.1 Highlighting Strings

Potion, like most programming languages, supports string literals like `"Hello, world!"`. We should highlight these as strings. To do this we'll use the `syntax region` command. Add the following to your Potion syntax file:

```
syntax region potionString start=/\v"/ skip=/\v\\./ end=/\v"/
highlight link potionString String
```

Close and reopen your `factorial.pn` file and you'll see that the string at the end of the file is highlighted!

The last line here should be familiar. Reread the previous two chapters if you don't understand what it does.

The first line adds to a syntax group using a "region". Regions have a "start" pattern and an "end" pattern that specify where they start and end. In this case, a Potion string starts when we see a double quote and ends when we see the next double quote.

The "skip" argument to `syntax region` allows us to handle string escapes like `"She said: \"Vimscript is tricky, but useful\"!"`.

If we didn't use the `skip` argument Vim would end the string at the `"` before `Vimscript`, which is not what we want!

In a nutshell, the `skip` argument to `syntax region` tells Vim: "once you start matching this region, I want you to ignore anything that matches `skip`, even if it would normally be considered the end of the region".

Take a few minutes and think through this. What happens with something like `"foo \\" bar"`? Is that the correct behavior? Will that *always* be the correct behavior? Close this book, take a few minutes and really *think* about this!

47.2 Exercises

Add syntax highlighting for single quoted strings.

Read `:help syn-region`.

Reading that should take longer than reading this chapter. Pour yourself a drink, you've earned it!

48 Basic Folding

If you've never used code folding in Vim, you don't know what you're missing. Read `:help usr_28` and spend some time playing around with it in your normal work. Come back to this chapter once you've got it in your fingers.

48.1 Types of Folding

Vim supports six different ways of defining how your text should be folded.

Manual

You create the folds by hand and they're stored in RAM by Vim. When you close Vim they go away and you have to recreate them the next time you edit the file.

This method can be handy if you combine it with some custom mappings to make it easy to create folds. We won't do that in this book, but keep it in the back of your mind in case you run across a case where it could be handy.

Marker

Vim folds your code based on characters in the actual text.

Usually these characters are put in comments (like `// {{{`), but in some languages you can get away with using something in the language's syntax itself, like `{` and `}` in Javascript files.

It may seem ugly to clutter up your code with comments that are purely for your text editor, but the advantage is that it lets you hand-craft folds for a specific file. This can be really nice if you're working with a large file that you want to organize in a very specific way.

Diff

A special folding mode used when diff'ing files. We won't talk about this one at all because Vim automatically handles it.

Expr

This lets you use a custom piece of Vimscript to define where folds occur. It's the most powerful method, but also requires the most work. We'll talk about this in the next chapter.

Indent

Vim uses your code's indentation to determine folds. Lines at the same indentation level fold together, and lines with only whitespace (and blank lines) are simply folded with their neighbors.

This is essentially free to use because your code is already indented; all you have to do is turn it on. This will be our first method of adding folding to Potion files.

48.2 Potion Folding

Let's take a look at our sample Potion file once again:

```
factorial = (n):
    total = 1
    n to 1 (i):
        total *= i.
    total.

10 times (i):
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.
```

The bodies of the function and loop are both indented. This means we can get some basic folding with very little effort by using indent folding.

Before we start, go ahead and add a comment above the `total *= i.` line so we have a nice multiple-line inner block to test with. You'll learn why we need to do this when you do the exercises, but for now just trust me. The file should now look like this:

```
factorial = (n):
    total = 1
    n to 1 (i):
        # Multiply the running total.
        total *= i.
    total.

10 times (i):
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.
```

Create an `ftplugin` folder in your Potion plugin's repository, and create a `potion` folder inside that. Finally, create a `folding.vim` file inside of *that*.

Remember that Vim will run the code in this file whenever it sets a buffer's `filetype` to `potion` (because it's in a folder named `potion`).

Putting all folding-related code into its own file is generally a good idea and will help us keep the various functionality of our plugin organized.

Add the following line to this file:

```
setlocal foldmethod=indent
```

Close Vim and open the `factorial.pn` file again. Play around with the new folding with `zR`, `zM`, and `za`.

One line of Vimscript gave us some useful folding! That's pretty cool!

You might notice that the lines inside the inner loop of the `factorial` function aren't folded even though they're indented. What's going on?

It turns out that by default Vim will ignore lines beginning with a `#` character when using `indent` folding. This works great when editing C files (where `#` signals a preprocessor directive) but isn't very helpful when you're editing other types of files.

Let's add one more line to the `ftplugin/potion/folding.vim` file to fix this:

```
setlocal foldmethod=indent
setlocal foldignore=
```

Close and reopen `factorial.pn` and now the inner block will be folded properly.

48.3 Exercises

Read `:help foldmethod`.

Read `:help fold-manual`.

Read `:help fold-marker` and `:help foldmarker`.

Read `:help fold-indent`.

Read `:help fdl` and `:help foldlevelstart`.

Read `:help foldminlines`.

Read `:help foldignore`.

49 Advanced Folding

In the last chapter we used Vim's `indent` folding to add some quick and dirty folding to Potion files.

Open `factorial.pn` and make sure all the folds are closed with `zM`. The file should now look something like this:

```
factorial = (n):
+--  5 lines: total = 1

10 times (i):
+--  4 lines: i string print
```

Toggle the first fold and it will look like this:

```
factorial = (n):
    total = 1
    n to 1 (i):
+---  2 lines: # Multiply the running total.
    total.

10 times (i):
+--  4 lines: i string print
```

This is pretty nice, but I personally prefer to fold the first line of a block with its contents. In this chapter we'll write some custom folding code, and when we're done our folds will look like this:

```
factorial = (n):
    total = 1
+---  3 lines: n to 1 (i):
    total.

+--  5 lines: 10 times (i):
```

This is more compact and (to me) easier to read. If you prefer the `indent` method that's okay, but do this chapter anyway just to get some practice writing Vim folding expressions.

49.1 Folding Theory

When writing custom folding code it helps to have an idea of how Vim "thinks" of folding. Here are the rules in a nutshell:

- Each line of code in a file has a "foldlevel". This is always either zero or a positive integer.
- Lines with a foldlevel of zero are *never* included in any fold.
- Adjacent lines with the same foldlevel are folded together.
- If a fold of level X is closed, any subsequent lines with a foldlevel greater than or equal to X are folded along with it until you reach a line with a level less than X.

It's easiest to get a feel for this with an example. Open a Vim window and paste the following text into it.

```
a
    b
    c
        d
        e
    f
g
```

Turn on `indent` folding by running the following command:

```
:setlocal foldmethod=indent
```

Play around with the folds for a minute to see how they behave.

Now run the following command to view the foldlevel of line 1:

```
:echom foldlevel(1)
```

Vim will display `0`. Now let's find the foldlevel of line 2:

```
:echom foldlevel(2)
```

Vim will display `1`. Let's try line 3:

```
:echom foldlevel(3)
```

Once again Vim displays `1`. This means that lines `2` and `3` are part of a level `1` fold.

Here are the foldlevels for each line:

```
a           0
    b       1
    c       1
        d   2
        e   2
    f       1
g           0
```

Reread the rules at the beginning of this section. Open and close each fold in this file, look at the foldlevels, and make sure you understand why the folds behave as they do.

Once you're confident that you understand how every line's foldlevel works to create the folding structure, move on to the next section.

49.2 First: Make a Plan

Before we dive into writing code, let's try to sketch out some rough "rules" for our folding.

First, lines that are indented should be folded together. We also want the *previous* line folded with them, so that something like this:

```
hello = (name):
    'Hello, ' print
    name print.
```

Will fold like this:

```
+--  3 lines: hello = (name):
```

Blank lines should be at the same level as *later* lines, so blank lines at the end of a fold won't be included in it. This means that this:

```
hello = (name):
    'Hello, ' print
    name print.

hello('Steve')
```

Will fold like this:

```
+--  3 lines: hello = ():

hello('Steve')
```

And *not* like this:

```
+--  4 lines: hello = ():
hello('Steve')
```

These rules are a matter of personal preference, but for now this is the way we're going to implement folding.

49.3 Getting Started

Let's get started on our custom folding code by opening Vim with two splits. One should contain our `ftplugin/potion/folding.vim` file, and the other should contain our sample `factorial.pn`.

In the previous chapter we closed and reopened Vim to make our changes to `folding.vim` take effect, but it turns out there's an easier way to do that.

Remember that any files inside `ftplugin/potion/` will be run whenever the `filetype` of a buffer is set to `potion`. This means you can simply run `:set ft=potion` in the split containing `factorial.pn` and Vim will reload the folding code!

This is much faster than closing and reopening the file every time. The only thing you need to remember is that you have to *save* `folding.vim` to disk, otherwise your unsaved changes won't be taken into account.

49.4 Expr Folding

We're going to use Vim's `expr` folding to give us unlimited flexibility in how our code is folded.

We can go ahead and remove the `foldignore` from `folding.vim` because it's only relevant when using `indent` folding. We also want to tell Vim to use `expr` folding, so change the contents of `folding.vim` to look like this:

```
setlocal foldmethod=expr
setlocal foldexpr=GetPotionFold(v:lnum)

function! GetPotionFold(lnum)
    return '0'
endfunction
```

The first line simply tells Vim to use `expr` folding.

The second line defines the expression Vim should use to get the foldlevel of a line. When Vim runs the expression it will set `v:lnum` to the line number of the line it wants to know about. Our expression will call a custom function with this number as an argument.

Finally we define a dummy function that simply returns `'0'` for every line. Note that it's returning a String and not an Integer. We'll see why shortly.

Go ahead and reload the folding code by saving `folding.vim` and running `:set ft=potion` in `factorial.pn`. Our function returns `'0'` for every line, so Vim won't fold anything at all.

49.5 Blank Lines

Let's take care of the special case of blank lines first. Modify the `GetPotionFold` function to look like this:

```
function! GetPotionFold(lnum)
    if getline(a:lnum) =~? '\v^\s*$'
        return '-1'
    endif

    return '0'
endfunction
```

We've added an `if` statement to take care of the blank lines. How does it work?

First we use `getline(a:lnum)` to get the content of the current line as a String.

We compare this to the regex `\v^\s*$`. Remember that `\v` turns on "very magic" ("sane") mode. This regex will match "beginning of line, any number of whitespace characters, end of line".

The comparison is using the case-insensitive match operator `=~?`. Technically we don't have to be worried about case since we're only matching whitespace, but I prefer to be more explicit when using comparison operators on Strings. You can use `=~` instead if you prefer.

If you need a refresher on using regular expressions in Vim you should go back and reread the "Basic Regular Expressions" chapter and the chapters on the "Grep Operator".

If the current line has some non-whitespace characters it won't match and we'll just return `'0'` as before.

If the current line *does* match the regex (i.e. if it's empty or just whitespace) we return the string `'-1'`.

Earlier I said that a line's foldlevel can be zero or a positive integer, so what's happening here?

49.6 Special Foldlevels

Your custom folding expression can return a foldlevel directly, or return one of a few "special" strings that tell Vim how to fold the line without directly specifying its level.

`'-1'` is one of these special strings. It tells Vim that the level of this line is "undefined". Vim will interpret this as "the foldlevel of this line is equal to the foldlevel of the line above or below it, whichever is smaller".

This isn't *exactly* what our plan called for, but we'll see that it's close enough and will do what we want.

Vim can "chain" these undefined lines together, so if you have two in a row followed by a line at level 1, it will set the last undefined line to 1, then the next to last to 1, then the first to 1.

When writing custom folding code you'll often find a few types of line that you can easily set a specific level for. Then you'll use `'-1'` (and some other special foldlevels we'll see soon) to "cascade" the proper folding levels to the rest of the file.

If you reload the folding code for `factorial.pn` Vim *still* won't fold any lines together. This is because all the lines have a foldlevel of either zero or "undefined". The level `0` will "cascade" through the undefined lines and eventually all the lines will have their foldlevel set to `0`.

49.7 An Indentation Level Helper

To tackle non-blank lines we'll need to know their indentation level, so let's create a small helper function to calculate it for us. Add the following function above `GetPotionFold`:

```
function! IndentLevel(lnum)
    return indent(a:lnum) / &shiftwidth
endfunction
```

Reload the folding code. Test out your function by running the following command in the `factorial.pn` buffer:

```
:echom IndentLevel(1)
```

Vim displays `0` because line `1` is not indented. Now try it on line `2`:

```
:echom IndentLevel(2)
```

This time Vim displays `1`. Line two has `4` spaces at the beginning, and `shiftwidth` is set to `4`, so `4` divided by `4` is `1`.

`IndentLevel` is fairly straightforward. The `indent(a:lnum)` returns the number of spaces at the beginning of the given line number. We divide that by the `shiftwidth` of the buffer to get the indentation level.

Why did we use `&shiftwidth` instead of just dividing by `4`? If someone prefers two-space indentation in their Potion files, dividing by `4` would produce an incorrect result. We use the `shiftwidth` setting to allow for any number of spaces per level.

49.8 One More Helper

It might not be obvious where to go from here. Let's stop and think about what type of information we need to have to figure out how to fold a non-blank line.

We need to know the indentation level of the line itself. We've got that covered with the `IndentLevel` function, so we're all set there.

We'll also need to know the indentation level of the *next non-blank line*, because we want to fold the "header" lines with their indented bodies.

Let's write a helper function to get the number of the next non-blank line after a given line. Add the following function above `IndentLevel`:

```
function! NextNonBlankLine(lnum)
    let numlines = line('$')
    let current = a:lnum + 1

    while current <= numlines
        if getline(current) =~? '\v\S'
            return current
        endif

        let current += 1
    endwhile

    return -2
endfunction
```

This function is a bit longer, but is pretty simple. Let's take it piece-by-piece.

First we store the total number of lines in the file with `line('$')`. Check out the documentation for `line()` to see how this works.

Next we set the variable `current` to the number of the next line.

We then start a loop that will walk through each line in the file.

If the line matches the regex `\v\S`, which means "match a character that's *not* a whitespace character", then it must be non-blank, so we should return its line number.

If the line doesn't match, we loop around to the next one.

If the loop gets all the way to the end of the file without ever returning, then there are *no* non-blank lines after the current line! We return `-2` if that happens to indicate this. `-2` isn't a valid line number, so it's an easy way to say "sorry, there's no valid result".

We could have returned `-1`, because that's not a valid line number either. I could have even picked `0`, since line numbers in Vim start at `1`! So why did I pick `-2`, which seems like a strange choice?

I chose `-2` because we're working with folding code, and `'-1'` (and `'0'`) is a special Vim foldlevel string.

When my eyes are reading over this file and I see a `-1` my brain immediately thinks "undefined foldlevel". The same is true with `0`. I picked `-2` here simply to make it obvious that it's *not* a foldlevel, but is instead an "error".

If this feels weird to you, you can safely change the `-2` to a `-1` or a `0`. It's just a coding style preference.

49.9 Finishing the Fold Function

This is turning out to be quite a long chapter, so let's wrap up the folding function. Change `GetPotionFold` to look like this:

```
function! GetPotionFold(lnum)
    if getline(a:lnum) =~? '\v^\s*$'
        return '-1'
    endif

    let this_indent = IndentLevel(a:lnum)
    let next_indent = IndentLevel(NextNonBlankLine(a:lnum))

    if next_indent == this_indent
        return this_indent
    elseif next_indent < this_indent
        return this_indent
```

```
    elseif next_indent > this_indent
        return '>' . next_indent
    endif
endfunction
```

That's a lot of new code! Let's step through it to see how it all works.

Blanks

First we have our check for blank lines. Nothing's changed there.

If we get past that check we know we're looking at a non-blank line.

Finding Indentation Levels

Next we use our two helper functions to get the indent level of the current line, and the indent level of the next non-blank line.

You might wonder what happens if `NextNonBlankLine` returns our error condition of `-2`. If that happens, `indent(-2)` will be run. Running `indent()` on a nonexistent line number will just return `-1`. Go ahead and try it yourself with `:echom indent(-2)`.

`-1` divided by any `shiftwidth` larger than `1` will return `0`. This may seem like a problem, but it turns out that it won't be. For now, don't worry about it.

Equal Indents

Now that we have the indentation levels of the current line and the next non-blank line, we can compare them and decide how to fold the current line.

Here's the `if` statement again:

```
if next_indent == this_indent
    return this_indent
elseif next_indent < this_indent
    return this_indent
elseif next_indent > this_indent
    return '>' . next_indent
endif
```

First we check if the two lines have the same indentation level. If they do, we simply return that indentation level as the foldlevel!

An example of this would be:

```
a
b
    c
    d
e
```

If we're looking at the line containing "c", it has an indentation level of `1`. This is the same as the level of the next non-blank line ("d"), so we return `1` as the foldlevel.

If we're looking at "a", it has an indentation level of `0`. This is the same as the level of the next non-blank line ("b"), so we return `0` as the foldlevel.

This case fills in two foldlevels in this simple example:

```
a       0
b       ?
    c   1
    d   ?
e       ?
```

By pure luck this also handles the special "error" case of the last line as well! Remember we said that `next_indent` will be `0` if our helper function returns `-2`.

In this example the line "e" has an indent level of `0`, and `next_indent` will also be set to `0`, so this case matches and returns `0`. The foldlevels now look like this:

```
a       0
b       ?
    c   1
    d   ?
e       0
```

Lesser Indent Levels

Once again, here's the `if` statement:

```
if next_indent == this_indent
    return this_indent
elseif next_indent < this_indent
    return this_indent
elseif next_indent > this_indent
    return '>' . next_indent
endif
```

The second part of the `if` checks if the indentation level of the next line is *smaller* than the current line. This would be like line "d" in our example.

If that's the case, we once again return the indentation level of the current line.

Now our example looks like this:

```
a       0
b       ?
    c   1
    d   1
e       0
```

You could, of course, combine these two cases with `||`, but I prefer to keep them separate to make it more explicit. You might feel differently. It's a style issue.

Again, purely by luck, this case handles the other possible "error" case of our helper function. Imagine that we have a file like this:

```
a
    b
    c
```

The first case takes care of line "b":

```
a       ?
    b   1
    c   ?
```

Line "c" is the last line, and it has an indentation level of `1`. The `next_indent` will be set to `0` thanks to our helper functions. The second part of the `if` matches and sets the foldlevel to the current indentation level, or `1`:

```
a       ?
    b   1
    c   1
```

This works out great, because "b" and "c" will be folded together.

Greater Indentation Levels

Here's that tricky `if` statement for the last time:

```
if next_indent == this_indent
    return this_indent
elseif next_indent < this_indent
    return this_indent
elseif next_indent > this_indent
    return '>' . next_indent
endif
```

And our example file:

```
a       0
b       ?
    c   1
    d   1
e       0
```

The only line we haven't figured out is "b", because:

- "b" has an indent level of `0`.
- "c" has an indent level of `1`.
- 1 is not equal to 0, nor is 1 less than 0.

The last case checks if the next line has a *larger* indentation level than the current one.

This is the case that Vim's `indent` folding gets wrong, and it's the entire reason we're writing this custom folding in the first place!

The final case says that when the next line is indented more than the current one, it should return a string of a `>` character and the indentation level of the *next* line. What the heck is *that*?

Returning a string like `>1` from the fold expression is another one of Vim's "special" foldlevels. It tells Vim that the current line should *open* a fold of the given level.

In this simple example we could have just returned the number, but we'll see why this is important shortly.

In this case line "b" will open a fold at level 1, which makes our example look like this:

```
a       0
b       >1
    c   1
    d   1
e       0
```

That's exactly what we want! Hooray!

49.10 Review

If you've made it this far you should feel proud of yourself. Even simple folding code like this can be tricky and mind bending.

Before we end, let's go through our original `factorial.pn` code and see how our folding expression fills in the foldlevels of its lines.

Here's `factorial.pn` for reference:

```
factorial = (n):
    total = 1
    n to 1 (i):
        # Multiply the running total.
        total *= i.
    total.

10 times (i):
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.
```

First, any blank lines' foldlevels will be set to undefined:

```
factorial = (n):
    total = 1
    n to 1 (i):
        # Multiply the running total.
        total *= i.
    total.
                                        undefined
10 times (i):
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.
```

Any lines where the next line's indentation is *equal* to its own are set to its own level:

```
factorial = (n):
    total = 1                           1
    n to 1 (i):
        # Multiply the running total.   2
        total *= i.
    total.
                                        undefined
10 times (i):
    i string print                      1
    '! is: ' print                      1
    factorial (i) string print          1
    "\n" print.
```

The same thing happens when the next line's indentation is *less* than the current line's:

```
factorial = (n):
    total = 1                           1
    n to 1 (i):
        # Multiply the running total.   2
        total *= i.                     2
    total.                              1
                                        undefined
10 times (i):
    i string print                      1
    '! is: ' print                      1
    factorial (i) string print          1
    "\n" print.                         1
```

The last case is when the next line's indentation is *greater* than the current line's. When that happens the line's foldlevel is set to *open* a fold of the *next* line's foldlevel:

```
factorial = (n):                        >1
    total = 1                           1
    n to 1 (i):                         >2
        # Multiply the running total.   2
        total *= i.                     2
    total.                              1
                                        undefined
10 times (i):                           >1
    i string print                      1
    '! is: ' print                      1
    factorial (i) string print          1
    "\n" print.                         1
```

Now we've got a foldlevel for every line in the file. All that's left is for Vim to resolve any undefined lines.

Earlier I said that undefined lines will take on the smallest foldlevel of either of their neighbors.

That's how Vim's manual describes it, but it's not entirely accurate. If that were the case, the blank line in our file would take foldlevel `1`, because both of its neighbors have a foldlevel of `1`.

In reality, the blank line will be given a foldlevel of `0`!

The reason for this is that we didn't just set the `10 times (i):` line to foldlevel `1` directly. We told Vim that the line *opens* a fold of level `1`. Vim is smart enough to know that this means the undefined line should be set to `0` instead of `1`.

The exact logic of this is probably buried deep within Vim's source code. In general Vim behaves pretty intelligently when resolving undefined lines against "special" foldlevels, so it will usually do what you want.

Once Vim's resolved the undefined line it has a complete description of how to fold each line in the file, which looks like this:

```
factorial = (n):                        1
    total = 1                           1
    n to 1 (i):                         2
        # Multiply the running total.   2
        total *= i.                     2
    total.                              1
                                        0
10 times (i):                           1
    i string print                      1
    '! is: ' print                      1
    factorial (i) string print          1
    "\n" print.                         1
```

That's it, we're done! Reload the folding code and play around with the fancy new folding in `factorial.pn`.

49.11 Exercises

Read `:help foldexpr`.

Read `:help fold-expr`. Pay particular attention to all the "special" strings your expression can return.

Read `:help getline`.

Read `:help indent()`.

Read `:help line()`.

Figure out why it's important that we use `.` to combine the `›` character with the number in our folding function. What would happen if you used `+` instead? Why?

We defined our helper functions as global functions, but that's not a good idea. Change them to be script-local functions.

Put this book down and go outside for a while to let your brain recover from this chapter.

50 Section Movement Theory

If you've never used Vim's section movement commands (`[[`, `]]`, `[]` and `][`) take a second and read the help for them now. Go ahead and read `:help section` as well.

Confused yet? That's okay, so was I the first time I read that stuff. We're going to take a quick detour from writing code to learn about how these movements work, and then in the next chapter we'll make our Potion plugin support them.

50.1 Nroff Files

The four "section movement" commands are conceptually meant to move around between "sections" of a file.

All of these commands are designed to work with nroff files[1] by default. Nroff is a language like LaTeX or Markdown – it's used to write text that will be reformatted later (it's actually the format used by UNIX man pages).

Nroff files use a certain set of "macros" to define "section headings". For example, here's an excerpt from the `awk` man page:

```
.SH NAME                                                    ***
awk \- pattern-directed scanning and processing language
.SH SYNOPSIS                                                ***
.B awk
[
.BI \-F
.I fs
]
[
.BI \-v
.I var=value
]
[
.I 'prog'
|
.BI \-f
.I progfile
]
[
.I file ...
```

[1] http://en.wikipedia.org/wiki/Nroff

```
]
.SH DESCRIPTION                                          ***
.I Awk
scans each input
.I file
for lines that match ...
```

The lines starting with `.SH` are section headings. I've marked them with ***. The four section movement commands will move your cursor between these section heading lines.

Vim considers any line starting with `.` and one of the nroff heading macros to be a section header, *even when you're not editing an nroff file*!

You can change the macros by changing the `sections` setting, but Vim still requires a period at the beginning of the line, and the macros must be pairs of characters, so that setting doesn't add enough flexibility for Potion files.

50.2 Braces

Section movement commands *also* look for one more thing: an opening or closing curly brace (`{` or `}`) as the first character on a line.

`[[` and `]]` look for opening braces, while `[]` and `][` look for closing braces.

This extra "hack" allows you to move between sections of C-like languages easily. However, these rules are always the same no matter what type of file you're in!

Put the following into a buffer:

```
Test            A B
Test

.SH Hello       A B

Test

{               A
Test
}                 B

Test

.H World        A B
```

```
Test
Test            A B
```

Now run `:set filetype=basic` to tell Vim that this is a BASIC file, and try the section movement comments.

The `[[` and `]]` commands will move between the lines marked `A`, while `[]` and `][` move between the lines marked `B`.

This shows us that Vim always uses these same two rules for section movement, even for languages where neither one makes sense (like BASIC)!

50.3 Exercises

Read `:help section` again, now that you know the story of section movement.

Read `:help sections` just for the fun of it.

51 Potion Section Movement

Now that we know how section movement works, let's remap the commands to work in a way that makes sense for Potion files.

First we need to decide what "section" should mean for a Potion file. There are two pairs of section movement commands, so we can come up with two "schemes" and our users can use the one they prefer.

Let's use the following two schemes to define where Potion sections start:

1. Any line following a blank line that contains non-whitespace as the first character, or the first line in the file.
2. Any line that contains non-whitespace as the first character, an equal sign somewhere inside the line, and ends with a colon.

Using a slightly-expanded version of our sample `factorial.pn` file, here's what these rules will consider to be section headers:

```
# factorial.pn                              1
# Print some factorials, just for fun.

factorial = (n):                            1 2
    total = 1

    n to 1 (i):
        total *= i.

    total.

print_line = ():                            1 2
    "-=-=-=-=-=-=-=-\n" print.

print_factorial = (i):                      1 2
    i string print
    '! is: ' print
    factorial (i) string print
    "\n" print.

"Here are some factorials:\n\n" print       1

print_line ()                               1
```

```
10 times (i):
    print_factorial (i).
print_line ()
```

Our first definition tends to be more liberal. It defines a section to be roughly a "top-level chunk of text".

The second definition is more restrictive. It defines a section to be (effectively) a function definition.

51.1 Custom Mappings

Create a `ftplugin/potion/sections.vim` file in your plugin's repo. This is where we'll put the code for section movement. Remember that this code will be run whenever a buffer's `filetype` is set to `potion`.

We're going to remap all four section movement commands, so go ahead and create a "skeleton" file:

```
noremap <script> <buffer> <silent> [[ <nop>
noremap <script> <buffer> <silent> ]] <nop>

noremap <script> <buffer> <silent> [] <nop>
noremap <script> <buffer> <silent> ][ <nop>
```

Notice that we use `noremap` commands instead of `nnoremap`, because we want these to work in operator-pending mode too. That way you'll be able to do things like `d]]` to "delete from here to the next section".

We make the mappings buffer-local so they'll only apply to Potion files and won't take over globally.

We also make them silent, because the user won't care about the details of how we move between sections.

51.2 Using a Function

The code for performing the section movements is going to be very similar for all of the various commands, so let's abstract it into a function that our mappings will call.

You'll see this strategy in a lot of Vim plugins that create a number of similar mappings. It's easier to read and maintain than stuffing all the functionality in to a bunch of mapping lines.

Change the `sections.vim` file to contain this:

```
function! s:NextSection(type, backwards)
endfunction

noremap <script> <buffer> <silent> ]]
        \ :call <SID>NextSection(1, 0)<cr>

noremap <script> <buffer> <silent> [[
        \ :call <SID>NextSection(1, 1)<cr>

noremap <script> <buffer> <silent> ][
        \ :call <SID>NextSection(2, 0)<cr>

noremap <script> <buffer> <silent> []
        \ :call <SID>NextSection(2, 1)<cr>
```

I used Vimscript's long line continuation feature here because the lines were getting a bit long for my taste. Notice how the backslash to escape long lines comes at the *beginning* of the second line. Read `:help line-continuation` for more information.

Notice that we're using `<SID>` and a script-local function to avoid polluting the global namespace with our helper function.

Each mapping simply calls `NextSection` with the appropriate arguments to perform the movement. Now we can start implementing `NextSection`.

51.3 Base Movement

Let's think about what our function needs to do. We want to move the cursor to the next "section", and an easy way to move the cursor somewhere is with the `/` and `?` commands.

Edit `NextSection` to look like this:

```
function! s:NextSection(type, backwards)
    if a:backwards
        let dir = '?'
    else
        let dir = '/'
    endif

    execute 'silent normal! ' . dir . 'foo' . "\r"
endfunction
```

Now the function uses the `execute normal!` pattern we've seen before to perform either `/foo` or `?foo`, depending on the value given for `backwards`. This is a good start.

Moving on, we're obviously going to need to search for something other than `foo`, and that pattern is going to depend on whether we want to use the first or second definition of section headings.

Change `NextSection` to look like this:

```
function! s:NextSection(type, backwards)
    if a:type == 1
        let pattern = 'one'
    elseif a:type == 2
        let pattern = 'two'
    endif

    if a:backwards
        let dir = '?'
    else
        let dir = '/'
    endif

    execute 'silent normal! ' . dir . pattern . "\r"
endfunction
```

Now we just need to fill in the patterns, so let's go ahead and do that.

51.4 Top Level Text Sections

Replace the first `let pattern = '...'` line with the following:

```
let pattern = '\v(\n\n^\S|%^)'
```

To understand how the regular expression works, remember the definition of "section" that we're implementing:

> Any line following a blank line that contains a non-whitespace as the first character, or the first line in the file.

The `\v` at the beginning simply forces "very magic" mode like we've seen several times before.

The remainder of the regex is a group with two options. The first, `\n\n^\S`, searches for "a newline, followed by a newline, followed by a non-whitespace character". This finds the first set of lines in our definition.

The other option is `%^`, which is a special Vim regex atom that means "beginning of file".

Now we're at a point where we can try out the first two mappings. Save `ftplugin/potion/sections.vim` and run `:set filetype=potion` in your sample Potion buffer. The `[[` and `]]` commands should work, but somewhat oddly.

51.5 Search Flags

You'll notice that when you move between sections your cursor gets placed on the blank line above the one we actually want to move to. Think about why this happens before reading on.

The answer is that we searched using `/` (or `?`) and by default Vim places your cursor at the beginning of matches. For example, when you run `/foo` your cursor will be placed on the `f` in `foo`.

To tell Vim to put the cursor at the end of the match instead of the beginning, we can use a search flag. Try searching in your Potion file like so:

```
/factorial/e
```

Vim will find the word `factorial` and move you to it. Press `n` a few times to move through the matches. The `e` flag tells Vim to put the cursor at the end of matches instead of the beginning. Try it in the other direction too:

```
?factorial?e
```

Let's modify our function to use a search flag to put our cursor on the other end of the matches for this section:

```
function! s:NextSection(type, backwards)
    if a:type == 1
        let pattern = '\v(\n\n^\S|%^)'
        let flags = 'e'
    elseif a:type == 2
        let pattern = 'two'
        let flags = ''
    endif

    if a:backwards
        let dir = '?'
    else
        let dir = '/'
    endif
```

```
    execute 'silent normal! ' . dir . pattern . dir . flags . "\r"
endfunction
```

We've changed two things here. First, we set a `flags` variable depending on the type of section movement. For now we only worry about the first type, which is going to need a flag of `e`.

Second, we've concatenated `dir` and `flags` to the search string. This will add `?e` or `/e` depending on which direction we're searching.

Save the file, switch back to your sample Potion file and run `:set ft=potion` to make the changes take effect. Now try `[[` and `]]` to see them working properly!

51.6 Function Definitions

It's time to tackle our second definition of "section", and luckily this one is much more straightforward than the first. Recall the definition we need to implement:

> Any line that contains a non-whitespace as the first character, an equal sign somewhere inside the line, and ends with a colon.

We can use a fairly simple regex to find these lines. Change the second `let pattern = '...'` line in the function to this:

```
let pattern = '\v^\S.*\=.*:$'
```

This regex should look much less frightening than the last one. I'll leave it as an exercise for you to figure out how it works – it's a pretty straightforward translation of our definition.

Save the file, run `:set filetype=potion` in `factorial.pn`, and try out the new `]]` and `[]` mappings. They should work as expected.

We don't need a search flag here because putting the cursor at the beginning of the match (the default) works just fine.

51.7 Visual Mode

Our section movement commands work great in normal mode, but we need to add a bit more to make them work in visual mode as well. First, change the function to look like this:

```
function! s:NextSection(type, backwards, visual)
    if a:visual
        normal! gv
    endif

    if a:type == 1
        let pattern = '\v(\n\n^\S|%^)'
        let flags = 'e'
    elseif a:type == 2
        let pattern = '\v^\S.*\=.*:$'
        let flags = ''
    endif

    if a:backwards
        let dir = '?'
    else
        let dir = '/'
    endif

    execute 'silent normal! ' . dir . pattern . dir . flags . "\r"
endfunction
```

Two things have changed. First, the function takes an extra argument so it knows whether it's being called from visual mode or not. Second, if it's called from visual mode we run `gv` to restore the visual selection.

Why do we need to do this? Let's try something that will make it clear. Visually select some text in any buffer and then run the following command:

```
:echom "hello"
```

Vim will display `hello` but the visual selection will also be cleared!

When running an ex mode command with `:` the visual selection is always cleared. The `gv` command reselects the previous visual selection, so this will "undo" the clearing. It's a useful command, and can be handy in your day-to-day work too.

Now we need to update the existing mappings to pass `0` in for the new `visual` argument:

```
noremap <script> <buffer> <silent> ]]
        \ :call <SID>NextSection(1, 0, 0)<cr>

noremap <script> <buffer> <silent> [[
        \ :call <SID>NextSection(1, 1, 0)<cr>

noremap <script> <buffer> <silent> ][
        \ :call <SID>NextSection(2, 0, 0)<cr>

noremap <script> <buffer> <silent> []
        \ :call <SID>NextSection(2, 1, 0)<cr>
```

Nothing too complex there. Now let's add the visual mode mappings as the final piece of the puzzle:

```
vnoremap <script> <buffer> <silent> ]]
        \ :<c-u>call <SID>NextSection(1, 0, 1)<cr>

vnoremap <script> <buffer> <silent> [[
        \ :<c-u>call <SID>NextSection(1, 1, 1)<cr>

vnoremap <script> <buffer> <silent> ][
        \ :<c-u>call <SID>NextSection(2, 0, 1)<cr>

vnoremap <script> <buffer> <silent> []
        \ :<c-u>call <SID>NextSection(2, 1, 1)<cr>
```

These mappings all pass `1` for the `visual` argument to tell Vim to reselect the last selection before performing the movement. They also use the `<c-u>` trick we learned about in the Grep Operator chapters.

Save the file, `:set ft=potion` in the Potion file and you're done! Give your new mappings a try. Things like `v]]` and `d[]` should all work properly now.

51.8 Why Bother?

This has been a long chapter for some seemingly small functionality, but you've learned and practiced a lot of useful things along the way:

- Using `noremap` instead of `nnoremap` to create mappings that work as movements and motions.
- Using a single function with several arguments to simplify creating related mappings.
- Building up functionality in a Vimscript function incrementally.

- Building up an `execute 'normal! ...'` string programmatically.
- Using simple searches to move around with regexes.
- Using special regex atoms like `%^` (beginning of file).
- Using search flags to modify how searches work.
- Handling visual mode mappings that need to retain the visual selection.

Go ahead and do the exercises (it's just a bit of `:help` reading) and then grab some ice cream. You've earned it after this chapter!

51.9 Exercises

Read `:help search()`. This is a useful function to know, but you can also use the flags listed with the `/` and `?` commands.

Read `:help ordinary-atom` to learn about more interesting things you can use in search patterns.

52 External Commands

Vim follows the UNIX philosophy of "do one thing well". Instead of trying to cram all the functionality you could ever want inside the editor itself, the right way to use Vim is to delegate to external commands when appropriate.

Let's add some interaction with the Potion compiler to our plugin to get our feet wet with external commands in Vim.

52.1 Compiling

First we'll add a command to compile and run the current Potion file. There are a number of ways to do this, but we'll simply use an external command for now.

Create a `potion/ftplugin/potion/running.vim` file in your plugin's repo. This is where we'll create the mappings for compiling and running Potion files.

```
if !exists("g:potion_command")
    let g:potion_command = "potion"
endif

function! PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction

nnoremap <buffer> <localleader>r :call PotionCompileAndRunFile()<cr>
```

The first chunk stores the command used to execute Potion in a global variable, if that variable isn't already set. We've seen this kind of check before.

This will allow users to override it if `potion` isn't in their `$PATH` by putting a line like `let g:potion_command = "/Users/sjl/src/potion/potion"` in their `~/.vimrc` file.

The last line adds a buffer-local mapping that calls a function we've defined above. Remember that because this file is in the `ftdetect/potion` directory it will be run every time a file's `filetype` is set to `potion`.

The real functionality is in the `PotionCompileAndRunFile()` function. Go ahead and save this file, open up `factorial.pn` and press `<localleader>r` to run the mapping and see what happens.

If `potion` is in your `$PATH`, the file should be run and you should see its output in your terminal (or at the bottom of the window if you're using a GUI Vim). If you get an error about the `potion`

command not being found, you'll need to set `g:potion_command` in your `~/.vimrc` file as mentioned above.

Let's take a look at how `PotionCompileAndRunFile()` function works.

52.2 Bang!

The `:!` command (pronounced "bang") in Vim runs external commands and displays their output on the screen. Try it out by running the following command:

```
:!ls
```

Vim should show you the output of the `ls` command, as well as a "Press ENTER or type command to continue" prompt.

Vim doesn't pass any input to the command when run this way. Confirm this by running:

```
:!cat
```

Type a few lines and you'll see that the `cat` command spits them back out, just as it normally would if you ran `cat` outside of Vim. Use Ctrl-D to finish.

To run an external command without the `Press ENTER or type command to continue` prompt, use `:silent !`. Run the following command:

```
:silent !echo Hello, world.
```

If you run this in a GUI Vim like MacVim or gVim, you won't see the `Hello, world.` output of the command.

If you run it in a terminal Vim, your results may vary depending on your configuration. You may need to run `:redraw!` to fix your screen after running a bare `:silent !`.

Note that this command is `:silent !` and not `:silent!` (see the space?)! Those are two different commands, and we want the former! Isn't Vimscript great?

Let's look back at the `PotionCompileAndRun()` function:

```
function! PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction
```

First we run a `silent !clear` command, which should clear the screen without a `Press ENTER...` prompt. This will make sure we only see the output of this run, which is helpful when you're running the same commands over and over.

The next line uses our old friend `execute` to build a command dynamically. The command it builds will look something like this:

```
!potion factorial.pn
```

Notice that there's no `silent` here, so the user will see the output of the command and will have to press enter to go back to Vim. This is what we want for this particular mapping, so we're all set.

52.3 Displaying Bytecode

The Potion compiler has an option that will let you view the bytecode it generates as it compiles. This can be handy if you're trying to debug your program at a very low level. Try it out by running the following command at a shell prompt:

```
potion -c -V factorial.pn
```

You should see a lot of output that looks like this:

```
-- parsed --
code ...
-- compiled --
; function definition: 0x109d6e9c8 ; 108 bytes
; () 3 registers
.local factorial ; 0
.local print_line ; 1
.local print_factorial ; 2
...
[ 2] move     1 0
[ 3] loadk    0 0   ; string
[ 4] bind     0 1
[ 5] loadpn   2 0   ; nil
[ 6] call     0 2
...
```

Let's add a mapping that will let the user view the bytecode generated for the current Potion file in a Vim split so they can easily navigate and examine it.

First, add the following line to the bottom of `ftplugin/potion/running.vim`:

```
nnoremap <buffer> <localleader>b :call PotionShowBytecode()<cr>
```

Nothing special there – it's just a simple mapping. Now let's sketch out the function that will do the work:

```
function! PotionShowBytecode()
    " Get the bytecode.

    " Open a new split and set it up.

    " Insert the bytecode.

endfunction
```

Now that we've got a little skeleton set up, let's talk about how to make it happen.

52.4 system()

There are a number of ways we could implement this, so I'll choose one that will come in handy later for you.

Run the following command:

```
:echom system("ls")
```

You should see the output of the `ls` command at the bottom of your screen. If you run `:messages` you'll see it there too. The `system()` Vim function takes a command string as a parameter and returns the output of that command as a String.

You can pass a second string as an argument to `system()`. Run the following command:

```
:echom system("wc -c", "abcdefg")
```

Vim will display `7` (with some padding). If you pass a second argument like this, Vim will write it to a temporary file and pipe it into the command on standard input. For our purposes we won't need this, but it's good to know.

Back to our function. Edit `PotionShowBytecode()` to fill out the first part of the skeleton like this:

```
function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))
    echom bytecode

    " Open a new split and set it up.

    " Insert the bytecode.

endfunction
```

Go ahead and try it out by saving the file, running `:set ft=potion` in `factorial.pn` to reload it, and using the `<localleader>b` mapping. Vim should display the bytecode at the bottom of the screen. Once you can see it's working you can remove the `echom` line.

52.5 Scratch Splits

Next we're going to open up a new split window for the user to show the results. This will let the user view and navigate the bytecode with all the power of Vim, instead of just reading it once from the screen.

To do this we're going to create a "scratch" split: a split containing a buffer that's never going to be saved and will be overwritten each time we run the mapping. Change the `PotionShowBytecode()` function to look like this:

```
function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))

    " Open a new split and set it up.
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.

endfunction
```

These new command should be pretty easy to follow.

`vsplit` creates a new vertical split for a buffer named `__Potion_Bytecode__`. We surround the name with underscores to make it clearer to the user that this isn't a normal file (it's a buffer just to hold the output). The underscores aren't special, they're just a convention.

Next we delete everything in this buffer with `normal! ggdG`. The first time the mapping is run this won't do anything, but subsequent times we'll be reusing the `__Potion_Bytecode__` buffer, so this clears it.

Next we prepare the buffer by setting two local settings. First we set its filetype to `potionbytecode`, just to make it clear what it's holding. We also change the `buftype` setting to `nofile`, which tells Vim that this buffer isn't related to a file on disk and so it should never try to write it.

All that's left is to dump the bytecode that we saved into the `bytecode` variable into this buffer. Finish off the function by making it look like this:

```
function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%") . " 2\
>&1")

    " Open a new split and set it up.
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.
    call append(0, split(bytecode, '\v\n'))
endfunction
```

The `append()` Vim function takes two arguments: a line number to append after, and a list of Strings to append as lines. For example, try running the following command:

```
:call append(3, ["foo", "bar"])
```

This will append two lines, `foo` and `bar`, below line `3` in your current buffer. In this case we're appending below line `0`, which means "at the top of the file".

We need a list of Strings to append, but we just have a single string with newline characters embedded in it from when we used `system()`. We use Vim's `split()` function to split that giant hunk of text into a list of Strings. `split()` takes a String to split and a regular expression to find the split points. It's pretty simple.

Now that the function is complete, go ahead and try out the mapping. When you run `<localleader>b` in the `factorial.pn` buffer Vim will open a new buffer containing the Potion bytecode. Play around with it by changing the source, saving the file, and running the mapping again to see the bytecode change.

52.6 Exercises

Read `:help bufname`.

Read `:help buftype`.

Read `:help append()`.

Read `:help split()`.

Read `:help :!`.

Read `:help :read` and `:help :read!` (we didn't cover these commands, but they're extremely useful).

Read `:help system()`.

Read `:help design-not`.

Currently our mappings require that the user save the file themselves before running the mapping in order for their changes to take effect. Undo is cheap these days, so edit the functions we wrote to save the current file for them.

What happens when you run the bytecode mapping on a Potion file with a syntax error? Why does that happen?

Change the `PotionShowBytecode()` function to detect when the Potion compiler returns an error, and show an error message to the user.

52.7 Extra Credit

Each time you run the bytecode mapping a new vertical split will be created, even if the user hasn't closed the previous one. If the user doesn't bother closing them they could end up with many extra windows stacked up.

Change `PotionShowBytecode()` to detect with a window is already open for the `__Potion_Bytecode__` buffer, and when that's the case switch to it instead of creating a new split.

You'll probably want to read `:help bufwinnr()` for this one.

52.8 More Extra Credit

Remember how we set the `filetype` of the temporary buffer to `potionbytecode`? Create a `syntax/potionbytecode.vim` file and define syntax highlighting for Potion bytecode buffers to make them easier to read.

53 Autoloading

We've written a fair amount of functionality for our Potion plugin, and that's all we're going to do in this book. Before we finish we'll talk about a few more important ways to polish it up and really make it shine.

First on the list is making our plugin more efficient with autoloading.

53.1 How Autoload Works

Currently when a user loads our plugin (by opening a Potion file) *all* of its functionality is loaded. Our plugin is still small so this probably isn't a big deal, but for larger plugins loading all of their code can take a noticeable amount of time.

Vim's solution to this is something called "autoload". Autoload lets you delay loading code until it's actually needed. You'll take a slight performance hit overall, but if your users don't always use every single bit of code in your plugin autoloading can be a huge speedup.

Here's how it works. Look at the following command:

```
:call somefile#Hello()
```

When you run this command, Vim will behave a bit differently than a normal function call.

If this function has already been loaded, Vim will simply call it normally.

Otherwise Vim will look for a file called `autoload/somefile.vim` in your `~/.vim` directory (and any Pathogen bundles).

If this file exists, Vim will load/source the file. It will then try to call the function normally.

Inside this file, the function should be defined like this:

```
function somefile#Hello()
    " ...
endfunction
```

You can use multiple `#` characters in the function name to represent subdirectories. For example:

```
:call myplugin#somefile#Hello()
```

This will look for the autoloaded file at `autoload/myplugin/somefile.vim`. The function inside it needs to be defined with the full autoload path:

```
function myplugin#somefile#Hello()
    " ...
endfunction
```

53.2 Experimenting

To get a feel for how this works, let's give it a try. Create a `~/.vim/autoload/example.vim` file and add the following to it:

```
echom "Loading..."

function! example#Hello()
    echom "Hello, world!"
endfunction

echom "Done loading."
```

Save the file and run `:call example#Hello()`. Vim will output the following:

```
Loading...
Done loading.
Hello, world!
```

This little demonstration proves a few things:

1. Vim really does load the `example.vim` file on the fly. It didn't even exist when we opened Vim, so it couldn't have been loaded on startup!
2. When Vim finds the file it needs to autoload, it loads the entire file before actually calling the function.

Without closing Vim, change the definition of the function to look like this:

```
echom "Loading..."

function! example#Hello()
    echom "Hello AGAIN, world!"
endfunction

echom "Done loading."
```

Save the file and **without closing Vim** run `:call example#Hello()`. Vim will simply output:

```
Hello, world!
```

Vim already has a definition for `example#Hello`, so it doesn't need to reload the file, which means:

1. The code outside the function wasn't run again.
2. It didn't pick up the changes to the function.

Now run `:call example#BadFunction()`. You'll see the loading messages again, as well as an error about a nonexistent function. But now try running `:call example#Hello()` again. This time you'll see the updated message!

By now you should have a pretty clear grip on what happens when Vim encounters a call to a function with an autoload-style name:

1. It checks to see if it has a function by that name defined already. If so, just call it.
2. Otherwise, find the appropriate file (based on the name) and source it.
3. Then attempt to call the function. If it works, great. If it fails, just print an error.

If that's not completely solid in your mind yet, go back and work through this demonstration again and try to see where each rule takes effect.

53.3 What to Autoload

Autoloading isn't free. There's some (small) overhead involved with setting it up, not to mention the ugly function names you need to sprinkle through your code.

With that said, if you're creating a plugin that won't be used *every* time a user opens a Vim session it's probably a good idea to move as much functionality into autoloaded files as possible. This will reduce the impact your plugin has on your users' startup times, which is important as people install more and more Vim plugins.

So what kind of things can be safely autoloaded? The answer is basically anything that's not directly called by your users. Mappings and custom commands can't be autoloaded (because they wouldn't be available for the users to call), but many other things can be.

Let's look at our Potion plugin and see what we can autoload.

53.4 Adding Autoloading to the Potion Plugin

We'll start with the compile and run functionality. Remember that our `ftplugin/potion/running.vim` file looked like this at the end of the previous chapter:

```
if !exists("g:potion_command")
    let g:potion_command = "/Users/sjl/src/potion/potion"
endif

function! PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction

function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))

    " Open a new split and set it up.
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.
    call append(0, split(bytecode, '\v\n'))
endfunction

nnoremap <buffer> <localleader>r :call PotionCompileAndRunFile()<cr>
nnoremap <buffer> <localleader>b :call PotionShowBytecode()<cr>
```

This file is already only called when a Potion file is loaded, so it doesn't add to the overhead of Vim's startup in general. But there may be some users who simply don't need this functionality, so if we can autoload some of it we'll save them a few milliseconds every time they open a Potion file.

Yes, in this case the savings won't be huge. But I'm sure you can imagine a plugin with many thousands of lines of functions where the time required to load them would be more significant.

Let's get started. Create an `autoload/potion/running.vim` file in your plugin repo. Then move the two functions into it and adjust their names, so they look like this:

```
echom "Autoloading..."

function! potion#running#PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction

function! potion#running#PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))

    " Open a new split and set it up.
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.
    call append(0, split(bytecode, '\v\n'))
endfunction
```

Notice how the `potion#running` portion of the function names matches the directory and file name where they live. Now change the `ftplugin/potion/running.vim` file to look like this:

```
if !exists("g:potion_command")
    let g:potion_command = "/Users/sjl/src/potion/potion"
endif

nnoremap <buffer> <localleader>r
            \ :call potion#running#PotionCompileAndRunFile()<cr>

nnoremap <buffer> <localleader>b
            \ :call potion#running#PotionShowBytecode()<cr>
```

Save the files, close Vim, and open up your `factorial.pn` file. Try using the mappings to make sure they still work properly.

Make sure that you see the diagnostic `Autoloading...` message only the first time you run one of the mappings (you may need to use `:messages` to see it). Once you confirm that autoloading is working properly you can remove that message.

As you can see, we've left the `nnoremap` calls that map the keys. We can't autoload these because the user would have no way to initiate the autoloading if we did!

This is a common pattern you'll see in Vim plugins: most of their functionality will be held in autoloaded functions, with just `nnoremap` and `command` commands in the files that Vim loads every time. Keep it in mind whenever you're writing a non-trivial Vim plugin.

53.5 Exercises

Read `:help autoload`.

Experiment a bit and find out how autoloading variables behaves.

Suppose you wanted to programatically force a reload of an autoload file Vim has already loaded, without bothering the user. How might you do this? You may want to read `:help silent!`. Please don't ever do this in real life.

54 Documentation

Our Potion plugin has a bunch of useful functionality in it, but it won't be really useful to anyone unless we document it so they know what it can do!

Vim's own documentation is superb. It's not overly wordy, but it's extremely thorough. It's also inspired many plugin authors to document their own plugins very well, which has resulted in a wonderful culture of strong documentation in the Vimscript community.

54.1 How Documentation Works

When you read a `:help` topic in Vim you've surely noticed that some things are highlighted differently than others. Let's take a look at how this works.

Open up any help topic (such as `:help help`) and run `:set filetype?`. Vim will display `filetype=help`. Now run `:set filetype=text`, and you'll see that the highlighting goes away. `:set filetype=help` again and it will come back.

It turns out that Vim help files are simply syntax-highlighted text files like any other file format! This means you can write your own and get the same highlighting.

Create a file called `doc/potion.txt` in your plugin repo. Vim/Pathogen looks for files inside `doc` folders when indexing help topics, so this is where we'll write the help for our plugin.

Open this file in Vim and run `:set filetype=help` so you can see the syntax highlighting as you type.

54.2 Help Header

The format of help files is a matter of personal taste. With that said, I'll talk about one way to structure them that seems to be popular with the modern Vimscript community.

The first line of the file should contain the filename of the help file, followed by a one-line description of what the plugin does. Add the following as the first line of your `potion.txt` file:

```
*potion.txt* functionality for the potion programming language
```

Surrounding a word with asterisks in a help file creates a "tag" that can be jumped to. Run `:Helptags` to tell Pathogen to rebuild the index of help tags, and then open a new Vim window and run `:help potion.txt`. Vim will open your help document like any other one.

Next you should put the title of your plugin along with a longer description. Some authors (including me) like to have a bit of fun with this and use some ASCII art to spice things up. Add a nice title section to the `potion.txt` file:

```
*potion.txt* functionality for the potion programming language

                  ___       _   _                  ~
                 / _ \___ | |_(_) ___  _ __   ~
                / /_)/ _ \| __| |/ _ \| '_ \  ~
               / ___/ (_) | |_| | (_) | | | | ~
               \/    \___/ \__|_|\___/|_| |_| ~

          Functionality for the Potion programming language.
        Includes syntax highlighting, code folding, and more!
```

I got those fun letters by running the `figlet -f ogre "Potion"` command. Figlet[1] is a great little program for generating ASCII art text. The ~ characters at the end of the lines ensure that Vim doesn't try to highlight or hide individual characters inside the art.

54.3 What to Document

Next usually comes a table of contents. First, though, let's decide what we actually want to document.

When writing documentation for a new plugin I usually start with the following list of sections and work from there:

- Introduction
- Usage
- Mappings
- Configuration
- License
- Bugs
- Contributing
- Changelog
- Credits

If the plugin is large and requires an "overview" I'll write an introductory section that summarizes how things work. Otherwise I'll skip that and just move on.

A "usage" section should explain in, general, how the user will actually *use* your plugin. If they'll interact with it through mappings, tell them that. If there aren't too many mappings you can simply list them here, otherwise you may want to create a separate "mappings" section that lists them all.

[1] http://www.figlet.org/

The "configuration" section should list each and every user-modifiable variable, along with its effects and its default value.

The "license" section should specify what license the plugin's code is under, as well as a URL where the user can find the full text of that license. Don't include the full text in the actual help file – most users know what the common licenses mean and it just clutters things up.

The "bugs" section should be short and sweet. List any major bugs that you're aware of but haven't gotten around to fixing, and tell the user how they can report new bugs they find to you.

If you want your users to be able to contribute bug fixes and features for the plugin back to you, they'll need to know how to do it. Should they send a pull request on GitHub? On Bitbucket? Send a patch in an email? Any/all of the above? Include a "contributing" section makes it clear how you prefer to receive code.

A changelog is a wonderful thing to include so that when users update your plugin from version X to version Y they can immediately see what changed. Also, I highlight recommend you pick a sane versioning scheme like Semantic Versioning[2] for your plugin and stick to it. Your users will thank you.

Finally, I like to include a "credits" section to mention my own name, list other plugins that this one was inspired by, thank contributors, and so on.

This is usually a good starting point. For more unique plugins you may feel the need to deviate from this list, and that's completely fine. There are no hard and fast rules except the following:

- Be thorough.
- Don't be too wordy.
- Take the user on a journey from having no idea what your plugin is to being an expert user of it.

54.4 Table of Contents

Now that we have an idea of what sections we'll include, add the following to the `potion.txt` file:

[2] http://semver.org/

```
==========================================================================
CONTENTS                                                  *PotionContents*

    1. Usage ................ |PotionUsage|
    2. Mappings ............. |PotionMappings|
    3. License .............. |PotionLicense|
    4. Bugs ................. |PotionBugs|
    5. Contributing ......... |PotionContributing|
    6. Changelog ............ |PotionChangelog|
    7. Credits .............. |PotionCredits|
```

There are a couple things to note here. First, the line of = characters will be syntax highlighted. You can use these lines to visually divide up sections of your help document. You can also use lines of - characters for subsections, if you want.

The `*PotionContents*` will create another tag, so a user can type `:help PotionContents` to go directly to the table of contents.

Each of the words surrounded by `|` characters creates a link to a tag. Users can place their cursor on the word in the help file and press `<c-]>` to jump to the tag, just as if they had typed `:help TheTag`. They can also click them with their mouse.

Vim will hide these `*` and `|` characters and syntax highlight them, so the result will be a nice, pretty table of contents people can use to get to what they're looking for.

54.5 Sections

You can create section headers like this:

```
==========================================================================
Section 1: Usage                                          *PotionUsage*

This plugin with automatically provide syntax highlighting for
Potion files (files ending in .pn).

It also...
```

Make sure to create the correct tags with the `*` characters so that all the links in your table of contents work properly.

Go ahead and create headers for each section in the table of contents.

54.6 Examples

I could try to go over all the syntax of help files and how to use it, but it's really a matter of taste. So instead I'll give you a list of several Vim plugins with extensive documentation.

For each one, copy the raw source of the documentation into a Vim buffer and set its filetype to `help` to see how it renders. Switch back to `text` when you want to see how an effect was created.

You may find it useful to use your Vimscript skills create a key mapping to toggle the `help` and `text` filetypes for the current buffer.

- Clam[3], my own plugin for working with shell commands. It's a pretty short example that hits most of the sections I talked about.
- The NERD tree[4], a file navigation plugin by Scrooloose. Note the general structure, as well as how he summarizes the mappings in an easy-to-read list before explaining every one in detail.
- Surround[5], a plugin for handling "surrounding" characters by Tim Pope. Note the lack of a table of contents, the different style of section headers, and the table column headers. Figure out how these things were done, and decide if you like them. It's a matter of taste.
- Splice[6], my own plugin for resolving three-way merge conflicts in version control systems. Note how the lists of mappings are formatted, and how I used ASCII-art diagrams to explain layouts. Sometimes a picture really is worth a thousand words.

Remember that all of the vanilla Vim documentation can also be used as an example too. That should give you plenty to study and learn from.

54.7 Write!

Now that you've see how some other plugins structured and wrote their documentation, fill in the sections for your Potion plugin.

If you're not used to writing technical documentation this might be a challenge. Learning to write certainly isn't simple, but like any other skill it definitely requires practice, so just go to it! It doesn't need to be perfect and you can always improve it later.

Don't be afraid to write something you're not completely sure about and then throw it away and rewrite it later. Often just getting *something* in the buffer will get your mind in the mood to write. It'll still be in version control if you ever want it back any way.

[3] https://github.com/sjl/clam.vim/blob/master/doc/clam.txt
[4] https://github.com/scrooloose/nerdtree/blob/master/doc/NERD_tree.txt
[5] https://github.com/tpope/vim-surround/blob/master/doc/surround.txt
[6] https://github.com/sjl/splice.vim/blob/master/doc/splice.txt

A good way to start is to imagine you've got a friend who also uses Vim sitting next to you. They've never used your plugin before but are intrigued, and your goal is to turn them into an expert user of it. Thinking about explaining things to an actual human being will help keep you grounded and avoid getting too deeply technical before you've established a good overview of how things work.

If you're still stuck and feel like you're not up to the challenge of writing the documentation for a full plugin, try doing something smaller. Pick a mapping in your ~/.vimrc file and document it fully in a comment. Explain what it's for, how to use it, and how it works. For example, this is in my own ~/.vimrc file:

```
" "Uppercase word" mapping.
"
" This mapping allows you to press <c-u> in insert mode to convert the
" current word to uppercase.  It's handy when you're writing names of
" constants and don't want to use Capslock.
"
" To use it you type the name of the constant in lowercase.  While
" your cursor is at the end of the word, press <c-u> to uppercase it,
" and then continue happily on your way:
"
"                            cursor
"                            v
"     max_connections_allowed|
"     <c-u>
"     MAX_CONNECTIONS_ALLOWED|
"                            ^
"                            cursor
"
" It works by exiting out of insert mode, recording the current cursor
" location in the z mark, using gUiw to uppercase inside the current
" word, moving back to the z mark, and entering insert mode again.
"
" Note that this will overwrite the contents of the z mark.  I never
" use it, but if you do you'll probably want to use another mark.
inoremap <C-u> <esc>mzgUiw`za
```

It's much shorter than the documentation for a full plugin, but it's a good exercise that will help you practice writing. It's also very helpful for people reading your ~/.vimrc if you put it up on Bitbucket or GitHub.

54.8 Exercises

Write the documentation for each section of the Potion plugin.

Read `:help help-writing` for help about writing help.

Read `:help :left`, `:help :right`, and `:help :center` to learn about three useful commands for getting your ASCII art perfect.

55 Distribution

By now you have the Vimscript skills to make Vim plugins many other people might find useful. This chapter will go over getting your plugins online and making them easily available, as well as how to get the word out about them to potential users.

55.1 Hosting

The first thing you'll need to do is get your plugin online so other people can download it. The canonical place for Vim plugins to live is the scripts section of the Vim website[1].

You'll need a free account on the website. Once you've got one you can click the "Add Script" link and fill out the form. It should be pretty self explanatory.

A recent trend in the past few years has been to distribute plugins by hosting their repositories in a public place like Bitbucket or GitHub. The rise in popularity of this method can probably be attributed to two factors. First, Pathogen has made it trivial to keep each installed plugin's code in its own separate location. The rise of distributed version control systems like Mercurial and Git and public hosting sites like Bitbucket and GitHub has also probably had an impact.

Providing repositories is handy for people who keep their dotfiles in a version-controlled repository of their own. Mercurial users can use Mercurial's "subrepositories" to keep track of plugin versions, and Git users can use submodules (though only for other Git repos, unlike Mercurial's subrepos).

Having a full repository for each of your installed plugins also makes it easier to debug when something goes wrong with them. You can use blame, bisection, and any of the other tools your VCS provides available to figure out what's going on. It's also easier to contribute fixes back if you already have the repository on your machine.

Hopefully I've convinced you that you should also make your plugin's repository available publicly. It doesn't really matter which service you use, as long as the repository is available *somewhere*.

55.2 Documentation

You've already documented your plugin thoroughly in Vim's internal help format, but your job isn't quite over yet. You're still going to want to come up with a quick overview that summarizes a few things:

1. What is your plugin all about?
2. Why would the user want to use it?
3. Why is it better than competing plugins (if any)?

[1] http://www.vim.org/scripts/

4. What's the license?
5. A link to a pretty version of the full documentation, rendered by the vim-doc[2] website.

This should go in your README file (which will be displayed on the landing page of Bitbucket and GitHub repos), and you can use it as the description for the plugin's entry on Vim.org.

Including some screenshots is almost always a great idea. Being a text-only editor doesn't mean Vim doesn't have a user interface.

55.3 Publicity

Once you've got your plugin settled into all its various homes on the web: tell the world about it! You could share it with your followers on Twitter, post it to the /r/vim[3] section of Reddit, write a blog entry about it on your own personal website, and announce it on the Vim mailing list[4] for starters.

Whenever you release a creation of yours into the wild you're going to get some praise and some criticism. Don't let negative words get to you too much. Listen to what they say, but keep a thick skin and don't get too emotional when someone points out flaws (valid or otherwise) in your work. No one is perfect, and this is the Internet, so you'll need to be able to take some heat and shrug it off if you want to stay happy and motivated.

55.4 Exercises

Create an account on Vim.org if you don't already have one.

Look at the README files for some of your favorite plugins to see how they're structured and what kind of information they include.

[2] http://vim-doc.heroku.com/

[3] http://reddit.com/r/vim/

[4] http://www.vim.org/maillist.php

56 What Now?

If you've read up to this point and completed all the examples and exercises you now have a pretty solid grasp of the basics of Vimscript. Don't worry though, there's still *plenty* left to learn!

Here are a few ideas of topics to look into if you're hungry for more.

56.1 Color Schemes

In this book we added syntax highlighting for Potion files. The other side of the coin is the creation of custom color schemes that define what color to display each syntax element.

Color schemes in Vim are fairly straightforward, if a bit repetitive, to make. Read `:help highlight` to learn the basics. You may want to take a look at some of the built-in color schemes to see how they structure their files.

If you're up for a challenge, take a look at the source of my own Bad Wolf[1] color scheme to see how I've used Vimscript to make the definition and maintenance much easier for myself. Pay attention to the "palette" dictionary and the `HL` function that dynamically builds the `highlight` commands.

56.2 The Command Command

Many plugins allow the user to interact with them through key mappings and function calls, but some prefer to create Ex commands instead. For example, the Fugitive[2] plugin creates commands like `:Gbrowse` and `:Gdiff` and leaves it up to the user to decide how to call them.

Commands like this are created with the `:command` command. Read `:help user-commands` to learn how to make your own. You should know enough Vimscript by now that reading Vim's documentation is sufficient for learning about new commands.

56.3 runtimepath

In this book I've kind of glossed over how Vim decides which files to load by saying "just use Pathogen". Now that you know a decent amount of Vimscript you can read `:help runtimepath` and check out Pathogen's source code[3] to find out what's *really* happening under the hood.

[1] https://github.com/sjl/badwolf/blob/master/colors/badwolf.vim
[2] https://github.com/tpope/vim-fugitive
[3] https://github.com/tpope/vim-pathogen/blob/master/autoload/pathogen.vim

56.4 Omnicomplete

Vim offers a number of different ways to complete text (read `:help ins-completion` for an overview). Most are fairly simple, but the most powerful of them is "omnicomplete" which lets you call a custom Vimscript function to determine completions in just about any way you could possibly think of.

When you're ready to dive into the rabbit hole of omnicompletion you can start with `:help omnifunc` and `:help coml-omni` and follow the trail from there.

56.5 Compiler Support

In our Potion plugin we created some mappings to compile and run our Potion files. Vim offers much deeper support for interacting with compilers, including parsing compile errors and providing a nice list that lets you jump to the line of each error.

If you're interested in this you can dive in by reading through `:help quickfix.txt` in its entirety. However, I will warn you now that `errorformat` is *not* for the faint of heart!

56.6 Other Languages

This book has focused on Vimscript, but Vim also offers interfaces in several other languages, like Python, Ruby, and Lua. This means you can script Vim in a different language if you don't like Vimscript.

It's still good to know Vimscript for editing your `~/.vimrc`, and for understanding the API Vim presents in each language. But using an alternate language can be a great way to escape from the cruftiness of Vimscript, especially for large plugins.

If you want to learn more about scripting Vim in a particular language, check out the help documents for it:

- `:help Python`
- `:help Ruby`
- `:help Lua`
- `:help perl-using`
- `:help MzScheme`

56.7 Vim's Documentation

As a final parting note, here's a list of Vim help topics that are especially useful, informative, interesting, or just plain fun (in no particular order):

- `:help various-motions`
- `:help sign-support`
- `:help virtualedit`
- `:help map-alt-keys`
- `:help error-messages`
- `:help development`
- `:help tips`
- `:help 24.8`
- `:help 24.9`
- `:help usr_12.txt`
- `:help usr_26.txt`
- `:help usr_32.txt`
- `:help usr_42.txt`

56.8 Exercises

Go write a Vim plugin for something you've always wanted and share it with the world!