Learn Vimscript the hard way

Steve Losh (steve@stevelosh.com)

April 4, 2013

Chapter 1

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!

Chapter 2

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 first and come back to this book when you're done.

2.1 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.

Chapter 3

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.

3.1 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.

3.2 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).

3.3 From the help system

3.3.1 help echo

`:ec[ho] {expr1} ..` Echoes each `{expr1}`, with a space in between. The first `{expr1}` starts on a new line. Also see `:comment`. Use `"\n"` to start a new line. Use `"\r"` to move the cursor to the first column. Uses the highlighting set by the `:echohl` command. Cannot be followed by a comment. Example:

```
:echo "the value of 'shell' is" &shell
```

3.3.2 help echom

`:echom[sg] {expr1} ..` Echo the expression(s) as a true message, saving the message in the `message-history`. Spaces are placed between the arguments as with the `:echo` command. But unprintable characters are displayed, not interpreted. The parsing works slightly different from `:echo`, more like `:execute`. All the expressions are first evaluated and concatenated before echoing anything. The expressions must evaluate to a Number or String, a Dictionary or List causes an error. Uses the highlighting set by the `:echohl` command. Example:

```
:echomsg "It's a Zizzer Zazzer Zuzz, as you can plainly see."
```

See `:echo-redraw` to avoid the message disappearing when the screen is redrawn.

3.3.3 help messages

This is an (incomplete) overview of various messages that Vim gives:

```
Press ENTER or type command to continue
```

This message is given when there is something on the screen for you to read, and the screen is about to be redrawn:

- After executing an external command (e.g., `":!ls"` and `"="`).
- Something is displayed on the status line that is longer than the width of the window, or runs into the 'showcmd' or 'ruler' output.
- Press `<Enter>` or `<Space>` to redraw the screen and continue, without that key being used otherwise.
- Press ':' or any other Normal mode command character to start that command.
- Press 'k', `<Up>`, 'u', 'b' or 'g' to scroll back in the messages. This works the same way as at the |more-prompt|. Only works when 'compatible' is off and 'more' is on.
- Pressing 'j', 'f', 'd' or `<Down>` is ignored when messages scrolled off the top of the screen, 'compatible' is off and 'more' is on, to avoid that typing one 'j' or 'f' too many causes the messages to disappear.
- Press `<C-Y>` to copy (yank) a modeless selection to the clipboard register.
- Use a menu. The characters defined for Cmdline-mode are used.
- When 'mouse' contains the 'r' flag, clicking the left mouse button works like pressing `<Space>`. This makes it impossible to select text though.
- For the GUI clicking the left mouse button in the last line works like pressing `<Space>`.

{Vi: only ":" commands are interpreted}

If you accidentally hit `<Enter>` or `<Space>` and you want to see the displayed text then use `g<`. This only works when 'more' is set.

To reduce the number of hit-enter prompts: - Set 'cmdheight' to 2 or higher. - Add flags to 'shortmess'. - Reset 'showcmd' and/or 'ruler'.

If your script causes the hit-enter prompt and you don't know why, you may find the `v:scrollstart` variable useful.

Also see 'mouse'. The hit-enter message is highlighted with the `hl-Question` group.

```
-- More --
-- More -- SPACE/d/j: screen/page/line down, b/u/k: up, q: quit
```

This message is given when the screen is filled with messages. It is only given when the 'more' option is on. It is highlighted with the `hl-MoreMsg` group.

Type	effect

`<CR>` or `<NL>` or `j` or `<Down>`	one more line
d	down a page (half a screen)
`<Space>` or f or `<PageDown>`	down a screen
G	down all the way, until the hit-enter prompt
`<BS>` or k or `<Up>`	one line back `(*)`
u	up a page (half a screen) `(*)`
b or `<PageUp>`	back a screen `(*)`
g	back to the start `(*)`
q, `<Esc>` or CTRL-C	stop the listing
:	stop the listing and enter a command-line
`<C-Y>`	yank (copy) a modeless selection to the clipboard (`"*` and `"+` registers)
{menu-entry}	what the menu is defined to in Cmdline-mode.
`<LeftMouse> (**)`	next page

Any other key causes the meaning of the keys to be displayed.

`(*)` backwards scrolling is {not in Vi}. Only scrolls back to where messages started to scroll.

`(**)` Clicking the left mouse button only works:

```
- For the GUI: in the last line of the screen.
- When 'r' is included in 'mouse' (but then selecting text won't
  work).
```

Note:

The typed key is directly obtained from the terminal, it is not mapped and typeahead is ignored.

The `g<` command can be used to see the last page of previous command output. This is especially useful if you accidentally typed `<Space>` at the hit-enter prompt.

3.4 Exercises

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

Chapter 4

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.

4.1 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.

4.2 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.

4.3 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`.

4.4 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?
```

4.5 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.

4.6 From the help system

4.6.1 :help number

`number` `nu` `nonumber`nonu‘: boolean (default off), local to window

Print the line number in front of each line. When the `'n'` option is excluded from `'cpoptions'` a wrapped line will not use the column of line numbers (this is the default when `'compatible'` isn't set).

The `'numberwidth'` option can be used to set the room used for the line number. When a long, wrapped line doesn`'t start with the first character,'-'` characters are put before the number.

See `hl-LineNr` and `hl-CursorLineNr` for the highlighting used for the number.

`number_relativenumber`

The `relativenumber` option changes the displayed number to be relative to the cursor. Together with `'number'` there are these four combinations (cursor in line 3):

`'nonu'`	`'nu'`	`'nonu'`	`'nu'`
`'nornu'`	`'nornu'`	`'rnu'`	`'rnu'`
apple	1 apple	2 apple	2 apple
pear	2 pear	1 pear	1 pear
nobody	3 nobody	0 nobody	3 nobody
there	4 there	1 there	1 there

4.6.2 :help relativenumber

`'relativenumber'` `'rnu'` `'norelativenumber'` `'nornu'` boolean (default off), local to window

Show the line number relative to the line with the cursor in front of each line. Relative line numbers help you use the `count` you can precede some vertical motion commands (e.g. `j k + -`) with, without having to calculate it yourself. Especially useful in combination with other commands (e.g. `y d c < > gq gw =`).

When the `'n'` option is excluded from `'cpoptions'` a wrapped line will not

use the column of line numbers (this is the default when 'compatible' isn't set). The 'numberwidth' option can be used to set the room used for the line number.

When a long, wrapped line doesn't start with the first character, '-' characters are put before the number. See hl-LineNr and hl-CursorLineNr for the highlighting used for the number.

The number in front of the cursor line also depends on the value of 'number', see number_relativenumber for all combinations of the two options.

4.6.3 :help numberwidth

'numberwidth' 'nuw' 'numberwidth' number (Vim default: 4 Vi default: 8). local to window

Minimal number of columns to use for the line number. Only relevant when the 'number' or 'relativenumber' option is set or printing lines with a line number. Since one space is always between the number and the text, there is one less character for the number itself.

The value is the minimum width. A bigger width is used when needed to fit the highest line number in the buffer respectively the number of rows in the window, depending on whether 'number' or 'relativenumber' is set. Thus with the Vim default of 4 there is room for a line number up to 999. When the buffer has 1000 lines five columns will be used. The minimum value is 1, the maximum value is 10.

> NOTE:
>
> 'numberwidth' is reset to 8 when 'compatible' is set.

4.6.4 :help wrap

'wrap' 'nowrap' boolean (default on). local to window.

This option changes how text is displayed. It doesn't change the text in the buffer, see 'textwidth' for that.

When on, lines longer than the width of the window will wrap and displaying continues on the next line. When off lines will not wrap and only part of long lines will be displayed. When the cursor is moved to a part that is not shown, the screen will scroll horizontally.

The line will be broken in the middle of a word if necessary. See 'linebreak' to get the break at a word boundary.

To make scrolling horizontally a bit more useful, try this:

```
:set sidescroll=5
:set listchars+=precedes:<,extends:>
```

See 'sidescroll', 'listchars' and wrap-off. This option can't be set from a modeline when the 'diff' option is on.

4.6.5 :help shiftround

`'shiftround'` `'sr'` `'noshiftround'` `'nosr'` boolean (default off).

Round indent to multiple of `'shiftwidth'`. Applies to `>` and `<` commands. `CTRL-T` and `CTRL-D` in Insert mode always round the indent to a multiple of `'shiftwidth'` (this is Vi compatible).

> NOTE:
>
> This option is reset when `'compatible'` is set.

4.6.6 :help matchtime

`'matchtime'` `'mat'` number (default 5).

Tenths of a second to show the matching paren, when `'showmatch'` is set. Note that this is not in milliseconds, like other options that set a time. This is to be compatible with Nvi.

4.7 Exercises

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

Chapter 5

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.

5.1 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`.

5.2 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.

5.3 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.

Chapter 6

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.

6.1 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.

6.2 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.

6.3 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.

Chapter 7

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 \
```

7.1 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
```

7.2 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.

7.3 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.

7.4 From the help system

7.4.1 :help unmap

:unm[ap] {lhs}

:nun[map] {lhs}

:vu[nmap] {lhs}

:xu[nmap] {lhs}

:sunm[ap] {lhs}

:ou[nmap] {lhs}

:unm[ap]! {lhs}

:iu[nmap] {lhs}

:lu[nmap] {lhs}

:cu[nmap] {lhs}

Remove the mapping of {lhs} for the modes where the map command applies. The mapping may remain defined for other modes where it applies.

> Note:
>
> Trailing spaces are included in the {lhs}. This unmap does NOT work:

```
:map @@ foo
:unmap @@ | print
```

7.5 Exercises

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

Chapter 8

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?

8.1 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!

8.2 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.

8.3 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.

8.4 From the help system

8.4.1 :help mapleader

To define a mapping which uses the "mapleader" variable, the special string `"<Leader>"` can be used. It is replaced with the string value of "mapleader". If "mapleader" is not set or empty, a backslash is used instead. Example:

```
:map <Leader>A  oanother line<Esc>
```

Works like:

```
:map \A  oanother line<Esc>
```

But after:

```
:let mapleader = ","
```

It works like:

```
:map ,A  oanother line<Esc>
```

Note that the value of "mapleader" is used at the moment the mapping is defined. Changing "mapleader" after that has no effect for already defined mappings.

8.4.2 :help maplocalleader

`<LocalLeader>` is just like `<Leader>`, except that it uses "maplocalleader" instead of "mapleader". `<LocalLeader>` is to be used for mappings which are local to a buffer. Example:

```
  :map <buffer> <LocalLeader>A  oanother line<Esc>
```

In a global plugin `<Leader>` should be used and in a filetype plugin `<LocalLeader>`. "mapleader" and "maplocalleader" can be equal. Although, if you make them different, there is a smaller chance of mappings from global plugins to clash with mappings for filetype plugins. For example, you could keep "mapleader" at the default backslash, and set "maplocalleader" to an underscore.

8.5 Exercises

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.

Chapter 9

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))).

9.1 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.

9.2 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 "**s**ource my **v**imrc 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.

9.3 From the help system

9.3.1 :help myvimrc

Four places are searched for initializations. The first that exists is used, the others are ignored. The `$MYVIMRC` environment variable is set to the file that was first found, unless $MYVIMRC was already set and when using VIMINIT.

- The environment variable VIMINIT (see also |compatible-default|) `(*)`. The value of `$VIMINIT` is used as an Ex command line.
- The user vimrc file(s):

"$HOME/.vimrc"	(for Unix and OS/2) (*)
"$HOME/.vim/vimrc"	(for Unix and OS/2) (*)
"s:.vimrc"	(for Amiga) (*)
"home:.vimrc"	(for Amiga) (*)
"home:vimfiles:vimrc"	(for Amiga) (*)
"$VIM/.vimrc"	(for OS/2 and Amiga) (*)
"$HOME/_vimrc"	(for MS-DOS and Win32) (*)
"$HOME/vimfiles/vimrc"	(for MS-DOS and Win32) (*)
"$VIM/_vimrc"	(for MS-DOS and Win32) (*)

Note:

For Unix, OS/2 and Amiga, when ".vimrc" does not exist, "`_vimrc`" is also tried, in case an MS-DOS compatible file system is used. For MS-DOS and Win32 ".vimrc" is checked after "`_vimrc`", in case long file names are used.

Note:

For MS-DOS and Win32, "`$HOME`" is checked first. If no "`_vimrc`" or ".vimrc" is found there, "`$VIM`" is tried. See `$VIM` for when `$VIM` is not set. - The environment variable `EXINIT`. The value of `$EXINIT` is used as an Ex command line. - The user exrc file(s). Same as for the user vimrc file, but with "vimrc" replaced by "exrc". But only one of ".exrc" and "`_exrc`" is used, depending on the system. And without the (*)!

(*) Using this file or environment variable will cause 'compatible' to be off by default. See `compatible-default`.

9.4 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.

Chapter 10

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.

10.1 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.

10.2 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.

10.3 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.

10.4 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.

Chapter 11

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.

11.1 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.

11.2 From the help system

11.2.1 :help '<

To the last line or character of the last selected Visual area in the current buffer. For block mode it may also be the first character of the last line (to be able to define the block). Note that 'selection' applies, the position may be just after the Visual area.

11.2.2 :help L

To line [count] from bottom of window (default: Last line on the window) on the first non-blank character `linewise`. See also 'startofline' option. Cursor is adjusted for 'scrolloff' option.

11.2.3 :help H

To line [count] from top (Home) of window (default: first line on the window) on the first non-blank character |linewise|. See also 'startofline' option. Cursor is adjusted for 'scrolloff' option.

11.3 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.

Chapter 12

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.

12.1 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.

12.2 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.

Chapter 13

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.

13.1 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.)

13.2 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.

13.3 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.

13.4 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.

13.5 From the help system

13.5.1 :help local-options

Some of the options only apply to a window or buffer. Each window or buffer has its own copy of this option, thus can each have their own value. This allows you to set 'list' in one window but not in another. And set 'shiftwidth' to 3 in one buffer and 4 in another.

The following explains what happens to these local options in specific situations. You don't really need to know all of this, since Vim mostly uses the option values you would expect. Unfortunately, doing what the user expects is a bit complicated…

When splitting a window, the local options are copied to the new window. Thus right after the split the contents of the two windows look the same.

When editing a new buffer, its local option values must be initialized. Since the local options of the current buffer might be specifically for that buffer, these are not used. Instead, for each buffer-local option there also is a global value, which is used for new buffers. With `":set"` both the local and global value is changed. With `"setlocal"` only the local value is changed, thus this value is not used when editing a new buffer.

When editing a buffer that has been edited before, the last used window options are used again. If this buffer has been edited in this window, the values from

back then are used. Otherwise the values from the window where the buffer was edited last are used.

It's possible to set a local window option specifically for a type of buffer. When you edit another buffer in the same window, you don't want to keep using these local window options. Therefore Vim keeps a global value of the local window options, which is used when editing another buffer. Each window has its own copy of these values. Thus these are local to the window, but global to all buffers in the window. With this you can do:

```
:e one
:set list
:e two
```

Now the 'list' option will also be set in `"two"`, since with the `":set list"` command you have also set the global value.

```
:set nolist
:e one
:setlocal list
:e two
```

Now the 'list' option is not set, because `":set nolist"` resets the global value, `":setlocal list"` only changes the local value and `":e two"` gets the global value. Note that if you do this next:

```
:e one
```

You will get back the 'list' value as it was the last time you edited `"one"`. The options local to a window are remembered for each buffer. This also happens when the buffer is not loaded, but they are lost when the buffer is wiped out `:bwipe`.

13.5.2 :help setlocal

i:setl[ocal] ... Like `":set"` but set only the value local to the current buffer or window.

Not all options have a local value. If the option does not have a local value the global value is set. With the `"all"` argument: display local values for all local options. Without argument: Display local values for all local options which are different from the default.

When displaying a specific local option, show the local value.

For a global/local boolean option, when the global value is being used, `"--"` is displayed before the option name.

For a global option the global value is shown (but that might change in the future). {not in Vi}

:setl[ocal] {option} Set the local value of {option} to its global value by copying the value.

13.5.3 :help map-local

If the first argument to one of these commands is "<buffer>" the mapping will be effective in the current buffer only. Example:

```
:map <buffer>  ,w  /[.,;]<CR>
```

Then you can map ",w" to something else in another buffer:

```
:map <buffer>  ,w  /[#&!]<CR>
```

The local buffer mappings are used before the global ones. See <nowait> below to make a short local mapping not taking effect when a longer global one exists.

The "<buffer>" argument can also be used to clear mappings:

```
:unmap <buffer> ,w
:mapclear <buffer>
```

Local mappings are also cleared when a buffer is deleted, but not when it is unloaded. Just like local option values. Also see map-precedence.

Chapter 14

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.

14.1 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.

14.2 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.

14.3 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.

14.4 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".

14.5 From the help system

14.5.1 :help autocmd–events

You can specify a comma-separated list of event names. No white space can be used in this list. The command applies to all the events in the list.

For READING FILES there are four kinds of events possible:

```
BufNewFile                    starting to edit a non-existent file
BufReadPre BufReadPost          starting to edit an existing file
FilterReadPre FilterReadPost  read the temp file with filter output
FileReadPre FileReadPost        any other file read
```

Vim uses only one of these four kinds when reading a file. The "Pre" and "Post" events are both triggered, before and after reading the file.

Note that the autocommands for the `*ReadPre` events and all the Filter events are not allowed to change the current buffer (you will get an error message if this happens). This is to prevent the file to be read into the wrong buffer.

Note that the 'modified' flag is reset AFTER executing the BufReadPost and BufNewFile autocommands. But when the 'modified' option was set by the autocommands, this doesn't happen.

You can use the 'eventignore' option to ignore a number of events or all events.

Vim recognizes the following events. Vim ignores the case of event names (e.g., you can use "BUFread" or "bufread" instead of "BufRead").

Reading

`BufNewFile`	starting to edit a file that doesn't exist
`BufReadPre`	starting to edit a new buffer, before reading the file
`BufRead`	starting to edit a new buffer, after reading the file
`BufReadPost`	starting to edit a new buffer, after reading the file
`BufReadCmd`	before starting to edit a new buffer `Cmd-event`

FileReadPre	before reading a file with a ":read" command
FileReadPost	after reading a file with a ":read" command
FileReadCmd	before reading a file with a ":read" command Cmd-event
FilterReadPre	before reading a file from a filter command
FilterReadPost	after reading a file from a filter command
StdinReadPre	before reading from stdin into the buffer
StdinReadPost	After reading from the stdin into the buffer

Writing

BufWrite	starting to write the whole buffer to a file
BufWritePre	starting to write the whole buffer to a file
BufWritePost	after writing the whole buffer to a file
BufWriteCmd	before writing the whole buffer to a file Cmd-event
FileWritePre	starting to write part of a buffer to a file
FileWritePost	after writing part of a buffer to a file
FileWriteCmd	before writing part of a buffer to a file Cmd-event
FileAppendPre	starting to append to a file
FileAppendPost	after appending to a file
FileAppendCmd	before appending to a file Cmd-event
FilterWritePre	starting to write a file for a filter command or diff
FilterWritePost	after writing a file for a filter command or diff

Buffers

BufAdd	just after adding a buffer to the buffer list
BufCreate	just after adding a buffer to the buffer list
BufDelete	before deleting a buffer from the buffer list
BufWipeout	before completely deleting a buffer
BufFilePre	before changing the name of the current buffer
BufFilePost	after changing the name of the current buffer
BufEnter	after entering a buffer
BufLeave	before leaving to another buffer
BufWinEnter	after a buffer is displayed in a window

`BufWinLeave`	before a buffer is removed from a window
`BufUnload`	before unloading a buffer
`BufHidden`	just after a buffer has become hidden
`BufNew`	just after creating a new buffer
`SwapExists`	detected an existing swap file

Options

`FileType`	when the 'filetype' option has been set
`Syntax`	when the 'syntax' option has been set
`EncodingChanged`	after the 'encoding' option has been changed
`TermChanged`	after the value of 'term' has changed

Startup and exit

`VimEnter`	after doing all the startup stuff
`GUIEnter`	after starting the GUI successfully
`GUIFailed`	after starting the GUI failed
`TermResponse`	after the terminal response to `t_RV` is received
`QuitPre`	when using `:quit`, before deciding whether to quit
`VimLeavePre`	before exiting Vim, before writing the viminfo file
`VimLeave`	before exiting Vim, after writing the viminfo file

Various

`FileChangedShell`	Vim notices that a file changed since editing started
`FileChangedShellPost`	After handling a file changed since editing started
`FileChangedRO`	before making the first change to a read-only file
`ShellCmdPost`	after executing a shell command
`ShellFilterPost`	after filtering with a shell command
`FuncUndefined`	a user function is used but it isn't defined
`SpellFileMissing`	a spell file is used but it can't be found
`SourcePre`	before sourcing a Vim script
`SourceCmd`	before sourcing a Vim script `Cmd-event`
`VimResized`	after the Vim window size changed

`FocusGained`	Vim got input focus
`FocusLost`	Vim lost input focus
`CursorHold`	the user doesn't press a key for a while
`CursorHoldI`	the user doesn't press a key for a while in Insert mode
`CursorMoved`	the cursor was moved in Normal mode
`CursorMovedI`	the cursor was moved in Insert mode
`WinEnter`	after entering another window
`WinLeave`	before leaving a window
`TabEnter`	after entering another tab page
`TabLeave`	before leaving a tab page
`CmdwinEnter`	after entering the command-line window
`CmdwinLeave`	before leaving the command-line window
`InsertEnter`	starting Insert mode
`InsertChange`	when typing while in Insert or Replace mode
`InsertLeave`	when leaving Insert mode
`InsertCharPre`	when a character was typed in Insert mode, before inserting it
`ColorScheme`	after loading a color scheme
`RemoteReply`	a reply from a server Vim was received
`QuickFixCmdPre`	before a quickfix command is run
`QuickFixCmdPost`	after a quickfix command is run
`SessionLoadPost`	after loading a session file
`MenuPopup`	just before showing the popup menu
`CompleteDone`	after Insert mode completion is done
`User`	to be used in combination with ":doautocmd"

14.6 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!

Chapter 15

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.

15.1 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.

15.2 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.

Chapter 16

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.

16.1 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.

16.2 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.

16.3 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.

16.4 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.

16.5 From the help system

16.5.1 :help autocmd-groups

Autocommands can be put together in a group. This is useful for removing or executing a group of autocommands. For example, all the autocommands for syntax highlighting are put in the `"highlight"` group, to be able to execute `":doautoall highlight BufRead"` when the GUI starts.

When no specific group is selected, Vim uses the default group. The default group does not have a name. You cannot execute the autocommands from the default group separately; you can execute them only by executing autocommands for all groups.

Normally, when executing autocommands automatically, Vim uses the autocommands for all groups. The group only matters when executing autocommands with `":doautocmd"` or `":doautoall"`, or when defining or deleting autocommands.

The group name can contain any characters except white space. The group name `"end"` is reserved (also in uppercase).

The group name is case sensitive. Note that this is different from the event name!

‘:aug[roup] {name} Define the autocmd group name for the following ":autocmd" commands. The name "end" or "END" selects the default group.

`:aug[roup]! {name}` Delete the autocmd group {name}. Don't use this if there is still an autocommand using this group! This is not checked.

To enter autocommands for a specific group, use this method:

1. Select the group with ":augroup {name}".
2. Delete any old autocommands with ":au!".
3. Define the autocommands.
4. Go back to the default group with "augroup END".

Example:

```
:augroup uncompress
:  au!
:  au BufEnter *.gz    %!gunzip
:augroup END
```

This prevents having the autocommands defined twice (e.g., after sourcing the .vimrc file again).

16.6 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.

Chapter 17

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
```

17.1 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.

17.2 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.

17.3 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.

17.4 From the help system

17.4.1 :help omap-info

Operator-pending mappings can be used to define a movement command that can be used with any operator. Simple example: `":omap { w"` makes `"y{"` work like `"yw"` and `"d{"` like `"dw"`.

To ignore the starting cursor position and select different text, you can have the omap start Visual mode to select the text to be operated upon. Example that operates on a function name in the current line:

```
onoremap <silent> F :<C-U>normal! 0f(hviw<CR>
```

The CTRL-U (`<C-U>`) is used to remove the range that Vim may insert. The Normal mode commands find the first `'('` character and select the first word before it. That usually is the function name.

To enter a mapping for Normal and Visual mode, but not Operator-pending mode, first define it for all three modes, then unmap it for Operator-pending mode:

```
:map    xx something-difficult
:ounmap xx
```

Likewise for a mapping for Visual and Operator-pending mode or Normal and Operator-pending mode.

17.5 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.

Chapter 18

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>
```

18.1 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.

18.2 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.

18.3 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.

18.4 From the help system

18.4.1 :help normal

‘:norm[al][!] {commands} Execute Normal mode commands {commands}. This makes it possible to execute Normal mode commands typed on the command-line. {commands} are executed like they are typed. For undo all commands are undone together. Execution stops when an error is encountered.

If the [!] is given, mappings will not be used. Without it, when this command is called from a non-remappable mapping (`:noremap`), the argument can be mapped anyway.

{commands} should be a complete command. If {commands} does not finish a command, the last one will be aborted as if `<Esc>` or `<C-C>` was typed. This implies that an insert command must be completed (to start Insert mode, see `:startinsert`). A `":"` command must be completed as well. And you can’t use `"Q"` or `"gQ"` to start Ex mode.

The display is not updated while `":normal"` is busy.

{commands} cannot start with a space. Put a count of 1 (one) before it, `"1 "` is one space.

The ‘insertmode’ option is ignored for {commands}.

This command cannot be followed by another command, since any `'|'` is considered part of the command.

This command can be used recursively, but the depth is limited by ‘maxmapdepth’.

An alternative is to use `:execute`, which uses an expression as argument. This allows the use of printable characters to represent special characters.

Example:

```
:exe `"normal \`<c-w>`\`<c-w>`"`
```

`:{range}norm[al][!] {commands}` Execute Normal mode commands {commands} for each line in the {range}. Before executing the {commands}, the cursor is positioned in the first column of the range, for each line. Otherwise it’s the same as the `":normal"` command without a range.

18.4.2 :help execute

`:exe[cute] {expr1} ..` Executes the string that results from the evaluation of {expr1} as an Ex command. Multiple arguments are concatenated, with a space in between. To avoid the extra space use the `"."` operator to concatenate strings into one argument. {expr1} is used as the processed command, command line editing keys are not recognized. Cannot be followed by a comment. Examples:

```
:execute "buffer" nextbuf
:execute "normal" count . "w"
```

":execute" can be used to append a command to commands that don't accept a '|'. Example:

```
:execute '!ls' | echo "theend"
```

":execute" is also a nice way to avoid having to type control characters in a Vim script for a ":normal" command:

```
:execute "normal ixxx\<Esc>"
```

This has an <Esc> character, see |expr-string|.

Be careful to correctly escape special characters in file names. The fnameescape() function can be used for Vim commands, shellescape() for :! commands. Examples:

```
:execute "e " . fnameescape(filename)
:execute "!ls " . shellescape(expand('%:h'), 1)
```

Note: The executed string may be any command-line, but you cannot start or end a "while", "for" or "if" command. Thus this is illegal:

```
:execute 'while i > 5'
:execute 'echo "test" | break'
```

It is allowed to have a "while" or "if" command completely in the executed string:

```
:execute 'while i < 5 | echo i | let i = i + 1 | endwhile'
```

":execute", ":echo" and ":echon" cannot be followed by a comment directly, because they see the '"'as the start of a string. But, you can use'|' followed by a comment. Example:

```
:echo "foo" | "this is a comment
```

18.4.3 :help expr-quote

Note that double quotes are used.

A string constant accepts these special characters:

`\...`	three-digit octal number (e.g., `"\316"`)
`\..`	two-digit octal number (must be followed by non-digit)
`\.`	one-digit octal number (must be followed by non-digit)
`\x..`	byte specified with two hex numbers (e.g., `"\x1f"`
`\x.`	byte specified with one hex number (must be followed by non-hex char)
`\X..`	same as `\x..`
`\X.`	same as `\x.`
`\u....`	character specified with up to 4 hex numbers, stored according to the current value of 'encoding' (e.g., `"\u02a4"`)
`\U....`	same as `\u....`
`\b`	backspace `<BS>`
`\e`	escape `<Esc>`
`\f`	formfeed `<FF>`
`\n`	newline `<NL>`
`\r`	return `<CR>`
`\t`	tab `<Tab>`
`\\`	backslash
`\"`	double quote
`\<xxx>`	Special key named `"xxx"`. e.g. `"\<C-W>"` for CTRL-W. This is for use in mappings, the 0x80 byte is escaped. Don't use `<Char-xxxx>` to get a utf-8 character, use `\uxxxx` as mentioned above.

Note that `"\xff"` is stored as the byte 255, which may be invalid in some encodings. Use `"\u00ff"` to store character 255 according to the current value of 'encoding'.

Note that `"\000"` and `"\x00"` force the end of the string.

18.5 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.

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.

Contents