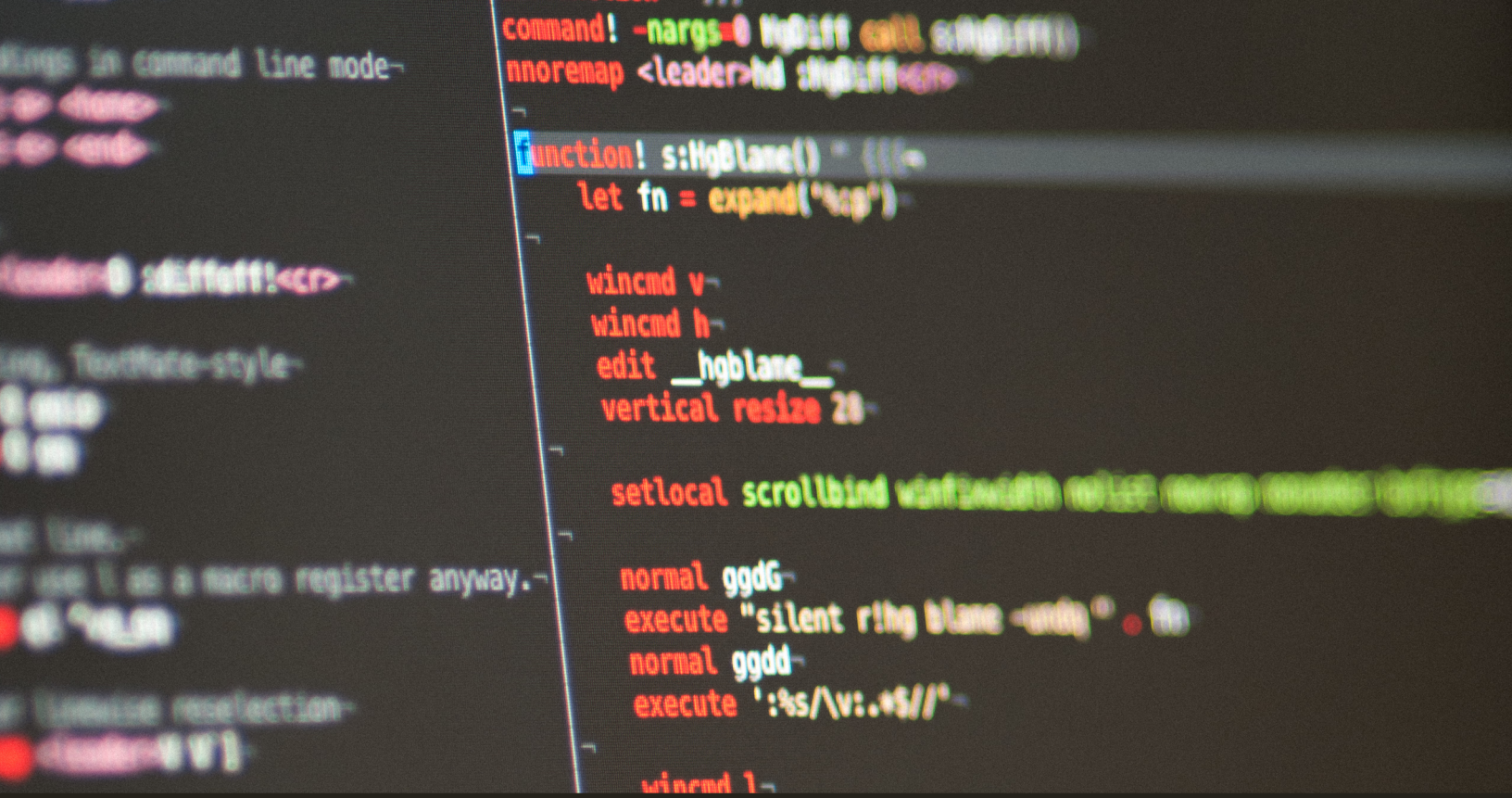
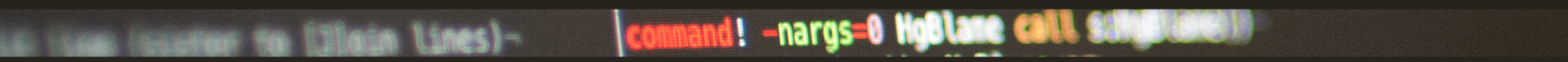
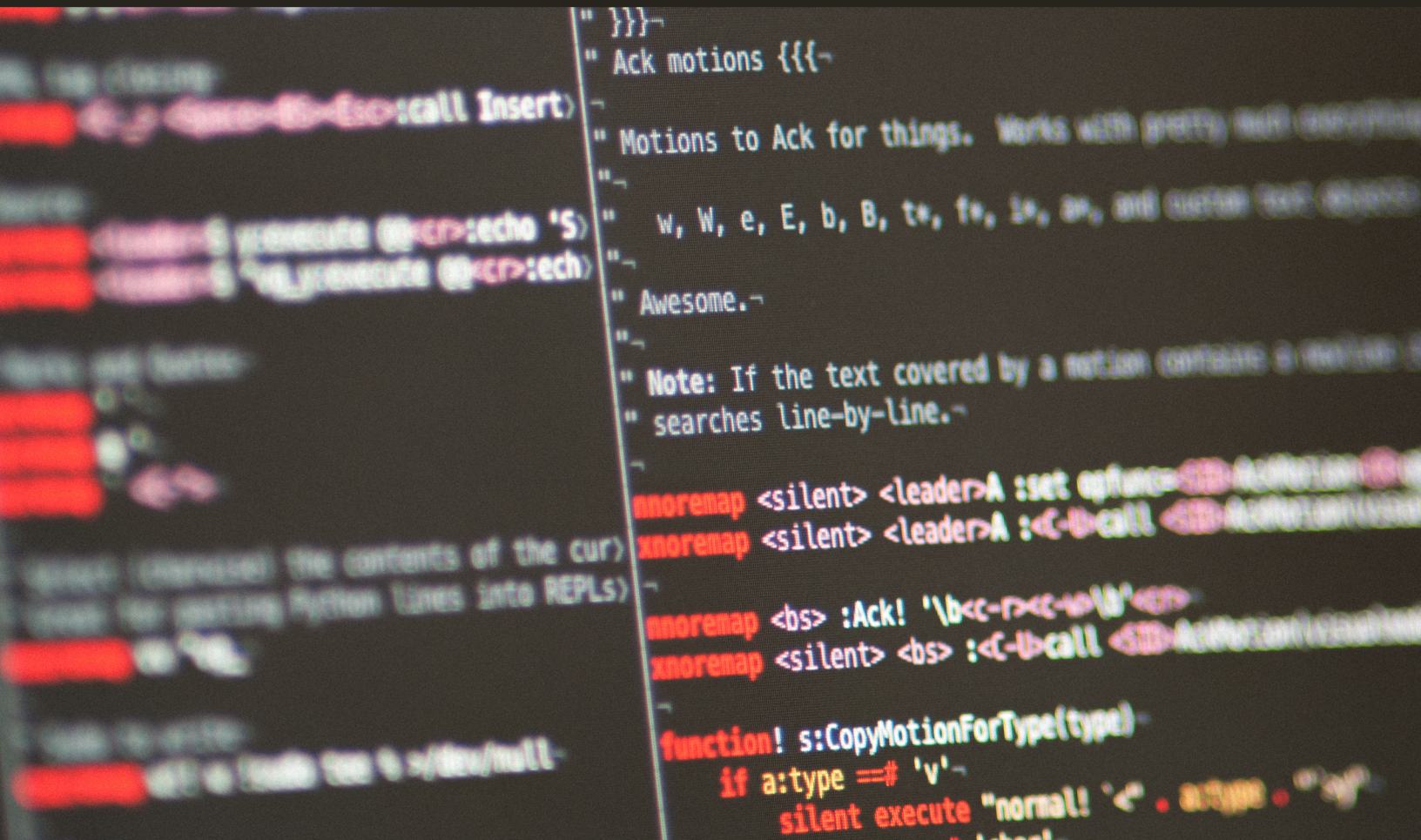


Learn Vimscript the Hard Way



Steve Losh



Learn Vimscript the Hard Way

Steve Losh

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

This version was published on 2013-04-04

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

Contents

Preface

Programmers shape ideas into text.

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

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

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

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

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

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

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

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

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

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

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

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

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

Good luck!

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

Creating a Vimrc File

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

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

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

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

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

Create this file if it doesn't already exist.

[1] http://learnpythonthehardway.org/

1 Echoing Messages

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

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

Try out `echo` by running the following command:

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

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

1.1 Persistent Echoing

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

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

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

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

```
:messages
```

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

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

1.2 Comments

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

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

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

1.3 Exercises

Read `:help echo`.

Read `:help echom`.

Read `:help messages`.

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

2 Setting Options

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

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

2.1 Boolean Options

Run the following command:

```
:set number
```

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

```
:set nonumber
```

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

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

2.2 Toggling Boolean Options

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

```
:set number!
```

The line numbers should reappear. Now run it again:

```
:set number!
```

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

2.3 Checking Options

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

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

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

2.4 Options with Values

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

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

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

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

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

2.5 Setting Multiple Options at Once

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

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

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

2.6 Exercises

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

Read `:help relativenumber`.

Read `:help numberwidth`.

Read `:help wrap`.

Read `:help shiftround`.

Read `:help matchtime`.

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

3 Basic Mapping

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

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

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

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

```
:map - x
```

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

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

```
:map - dd
```

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

3.1 Special Characters

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

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

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

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

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

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

3.2 Commenting

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

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

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

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

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

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

3.3 Exercises

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

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

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

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

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

4 Modal Mapping

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

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

Run this command:

```
:nmap \ dd
```

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

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

Run this command:

```
:vmap \ U
```

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

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

4.1 Muscle Memory

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

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

4.2 Insert Mode

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

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

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

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

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

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

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

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

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

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

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

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

4.3 Exercises

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

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

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

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

5 Strict Mapping

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

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

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

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

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

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

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

5.1 Recursion

Run this command:

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

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

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

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

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

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

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

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

```
:nunmap dd
```

5.2 Side Effects

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

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

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

5.3 Nonrecursive Mapping

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

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

Now press `\` and see what happens.

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

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

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

Always.

No, seriously, *always*.

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

5.4 Exercises

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

Read `:help unmap`.

6 Leaders

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

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

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

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

6.1 Mapping Key Sequences

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

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

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

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

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

6.2 Leader

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

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

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

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

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

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

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

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

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

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

6.3 Local Leader

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

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

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

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

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

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

6.4 Exercises

Read `:help mapleader`.

Read `:help maplocalleader`.

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

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

7 Editing Your Vimrc

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

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

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

That's not a typo. Read it again.

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

7.1 Editing Mapping

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

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

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

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

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

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

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

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

7.2 Sourcing Mapping

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

Let's add a mapping to make this easier:

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

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

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

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

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

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

7.3 Exercises

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

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

Read `:help myvimrc`.