Learning to play Vim

Master the best text editor, from Beginner to Composer

Matthieu Cneude

Contents

Rank I – Vim for Rookies

"Where am I? What am I doing here?"

This is your first thought as you wake up on a small beach surrounded by rocks. You were on a boat last night, with your billionaire friends drinking champagne and eating some caviar, as usual.

You don't remember what happened, and how you ended up here. You have a slight headache, amplified by the harsh sun directly above your head. Thankfully, you're still perfectly dressed, and perfectly dry, too.

It doesn't make much sense.

While looking around you, you catch the sight of a laptop bag abandoned on a rock. You reach it and open it quickly, wondering if there is an actual computer inside. There is! You might be able to contact somebody with this machine.

After the computer starts, a terminal run and opens a file, using a program called "Vim". Here's its content:

"Greating! Welcome to my Island! I hope you feel well.

I've followed you for quite a while now. Your lifestyle is so unhealthy! I don't like your clothes, either. But I don't care about that; what annoys me is your poor knowledge of the Glorious Vim.

That's right! You're here to become a Vim Composer, a rank which is only given to the best Vim players in the entire galaxy.

Why you? Why have you've been chosen for this incredible honor? Mainly because of your extreme intelligence and the beauty of your skin.

Now, go! Explore this island. Believe it or not, but you'll get better at Vim along the way. If I'm satisfied with your success, you might be able to come back to your old (and boring) life."

The file goes on, explaining the basics of Vim. There is a signature at the end: "The Island Master and The Carrier of The Word of Vim".

You feel lost. There is only the jungle in front of view, and behind the sea extends its quiet majesty till the horizon. What's happening here?

You sit down on the beach and you begin to read the file of this mysterious character, hoping to find some answers. It begins by a little table of content explaining what's ahead:

- The different modes of Vim: this is what make Vim so different from other editor, and so powerful too.
- How to move around with your precious keyboard.
- How to speak with Vim, again with your keyboard.

- How to undo and redo your changes.
- How to create Vim's configuration.

To try and experiment with what we'll see throughout this chapter, I advise you to open the file `rank_01/functions.lua` from the book companion.

Are you ready for the beginning of this adventure?

The Modes of Vim

Vim is a *modal editor*. It means that the keys of your keyboard will perform different actions depending on the mode you're in.

This is the first big difference with the text editors you're likely to be used to, and arguably one of the biggest reason why Vim is so beloved and powerful.

The modes you'll use the most are the NORMAL mode and the INSERT mode. This is what we'll discuss next; it's primordial to understand both of them to get what Vim is about.

The NORMAL Mode

Many text editors out there allow us to directly type some text with our beloved keyboards, as soon as they're running. What you type appear immediately on your screen. It's so usual that anybody starting Vim for the first time feels very confused.

That's right: Vim doesn't work like that.

To understand what I mean, try to open Vim. You'll see a welcome message. Now, if you try to type "x" in the editor, you might be surprised: no "x" appears on the damn screen! How come?

It's because Vim always start by default in NORMAL mode. This mode is not meant to *insert new content*, but to *edit already existing content*.

In NORMAL mode, we can use many different commands to move the cursor and target the exact content we want to edit. For example, instead of using the mouse to select the word you want to replace, you'll use your keyboard to do the same thing.

You can think of NORMAL mode commands as keystrokes, or keyboard shortcuts. Usually, in other text editors, you would need to hit a shortcut to perform an action, like selecting all your text with `CTRL-a`. Since you don't write anything in NORMAL mode, you have way more keystrokes to manipulate your text. You have access to all the keys on your keyboard, in fact.

But if there are so many NORMAL mode commands, how is it possible to remember them all? It's where it gets very interesting. These commands make sense in Vim, most of the time, compared to the usual and meaningless shortcuts you'll find in other editors.

Vim's NORMAL mode commands use mnemonics for you to remember them easily. Even better: they are *composable*: you can combine some of them in a logical, easy-to-remember way.

Let's take for example the shortcut `CTRL+shift+n` from a random IDE. By only looking at it, you've no idea what it does. In contrast, you'll see soon that it's possible to guess NORMAL mode commands in Vim.

Vim's NORMAL mode is a keyboard-centered way to control your editor, *telling* Vim what you want to do, for it to obey your mighty will.

To really grasp this concept, we need some text to edit. As we said, we write new text in NORMAL mode; so let's switch to the second most important mode, the INSERT mode.

The INSERT Mode

Let's now execute our first NORMAL mode command, which will switch Vim from NORMAL to INSERT mode. Simply hit `i` on your keyboard.

Depending on the editor you use (Vim or Neovim), and how your terminal is configured, the shape of your cursor might change. More importantly, you'll see the string `-- INSERT --` in the bottom left corner of Vim.

Welcome to INSERT mode.

In this mode, you're finally able to type the content you always wanted to bring to the world. Go ahead, don't be afraid: type anything you want, like you would do in more mainstream editors.

Now, let's try to hit the `ESC` key. The indicator `-- INSERT --` disappears.

Welcome back to NORMAL mode.

That's what do a Vim user, most of the time: juggling between NORMAL mode to edit existing content, and INSERT mode to insert new content.

The command `a` in NORMAL mode can also let you switch to INSERT mode, but it will do it **a**fter the character you're on. You can try it to see the difference.

That's what I'm talking about when I say that Vim uses mnemonic for the NORMAL mode commands: `i` for **i**nsert, `a` for insert **a**fter.

Let's summarize this knowledge:

`i`	Switch from NORMAL mode to INSERT mode (**i**nsert before current character).
`a`	Switch from NORMAL mode to INSERT mode (insert **a**fter current character).
`<Esc>`	Switch from INSERT mode to NORMAL mode.

There are more NORMAL mode commands allowing us to switch to INSERT mode. We'll see them in the next chapter; for now, the ones above should be enough for your editing needs.

The COMMAND-LINE Mode

The NORMAL mode and the INSERT mode are the ones you'll use most often. Following the order of importance, we'll find a third mode, the COMMAND-LINE mode.

Now, you might have noticed that we've spoken about NORMAL mode *commands*. What's this COMMAND-LINE mode? A new way to enter commands? Well, kind of.

In NORMAL mode, you run NORMAL mode commands; in COMMAND-LINE mode, you run *Ex commands*. This is quite confusing; that's why I'll speak about NORMAL mode keystrokes in this book, instead of NORMAL mode commands. That said, be aware that many other resources about Vim (including Vim's help) often speak about NORMAL mode commands.

So, what's the point of this COMMAND-LINE mode? You can run these special Ex commands allowing you to do many interesting things.

First, to switch to COMMAND-LINE mode, you need to use the NORMAL mode keystrokes `:`. Like in INSERT mode, to come back to NORMAL mode (our default, and soon favorite, mode), you need to hit the `ESC` key.

When you switch to COMMAND-LINE mode, your cursor moves automatically at the very bottom of Vim. Your cursor will be placed after a colon `:`, indicating that you can run Ex commands.

You can think of all these Ex commands as the menu you would NORMAL use in a text editor with a graphical user interface (GUI). In this mode, you can run Ex commands to save the file you're editing, substitute some text, and much, much more.

Here are some useful basic Ex commands you'll need to save your work, or simply quit Vim:

Command	Short Name	Description
`:write`	`:w`	Write (save) the current file open.
`:write!`	`:w!`	Write (save) the current file open (even if it's read-only).
`:edit {filepath}`	`:e {filepath}`	Edit the file with filepath `{filepath}`.
`:quit`	`:q`	Quit the current window.
`:quit!`	`:q!`	Quit the current window without saving.
`:wq`	`:wq`	Write (save) the current file and quit Vim.

A last important Ex command, maybe the most important of all: `:help {subject}`, to open the help about whatever `{subject}` you want.

This help is insanely complete. If you don't remember how to quit Vim for example, you can type `:help quit`.

I'll reference Vim's help very often in this book, in special blocks, at the end of some sections. They will allow you to dig deeper into the functionalities we'll cover.

For example:

Help Yourself

`:help vim-modes`

`:help write-quit`

Don't worry if you don't understand what's written in Vim's help, or if there is too much information. When you'll get more comfortable with Vim, it will make more sense. Simply bear with me for a little while.

A last important tip concerning the COMMAND-LINE mode: you can use the `TAB` key to complete Ex commands. Useful when you don't remember exactly the commands, or to discover new ones!

Like in a shell, you can also use the arrow keys `<up>` and `<down>` to go through your Ex command history.

Moving Around with Motions (NORMAL mode)

Let's now see how we can move our cursor horizontally or vertically, thanks to NORMAL mode keystrokes called *motions*.

Don't worry if you don't remember every single NORMAL mode keystroke we'll see here. You can always come back to this chapter and experiment with the one you forgot.

You can do so by going into the root of the repository and run `vim rank_01/functions.lua`.

Ditching the Arrow Keys

We're now at the most difficult part in our journey to learn Vim. At least it was the most difficult part for me: ditching the arrow keys to move the cursor.

As I said in the previous chapter, our fingers should stay on the row keys of the keyboard. First, for our typing speed and accuracy to improve, and second because the Vim's keystrokes you can use in NORMAL mode are more easily accessible from the row keys. Your hands shouldn't move too much; only your fingers should.

If you look at your arrow keys, you'll see that they're too far away from the row keys, forcing you to move your hands each time you want to use them. That's why, instead of using the arrow keys, many Vim users use the `h`, `j`, `k` and `l` keys instead, to move respectively left, down, up and right.

I would highly encourage you to use these keys, too. They'll improve your Vim experience significantly.

Why `hjkl`, and not some other keys close to the home row? For historic reasons. Vim is the ancestor of Vi, which was used on physical terminals. When you look at the keyboard of some of them, you'll see that the arrow keys are also the `hjkl` keys.

Like many other habits which seem ingrained in our brain, it will be difficult not to use the arrow keys at first. Your hand will come back to them over and over, even if you try not to. You need to accept this fact and be patient; you'll get there, and faster than you think.

For an easier transition, we can try to answer an important question: how to remember what `h`, `j`, `k` and `l` do in NORMAL mode? Here are some useful mnemonics:

- The `h` key is on the left of the sequence `hjkl`, and `l` is on the right. As a result, hitting `h` will move your cursor to the left, and `l` to the right.
- `j` moves your cursor down. Here are 3 mnemonics for this key:
  - With a bit of imagination, you can see `j` as an arrow going down.
  - The `j` has a little bump at the bottom of the key, meaning that it goes down.
  - Let's speak typography: `j` has a descender, meaning that part of the letter descends from its baseline. As a result, `j` goes down.
- `k` is the only letter left, so it has to go up. To come back to the secret art of typography, `k` is a letter with an ascender, meaning that part of the letter ascend from its baseline. As a result, `k` goes up.

Practice will get you there. I've got you covered for this one, with a revolutionary AAA game everybody will speak about in twenty years. To play it, you *must* use `hjkl`. If you prefer puzzle games, try this wonderful sokoban. You can use `hjkl` or the arrow keys this time, but try to only use `hjkl`.

Horizontal Motions

The keys `h` and `l` are not the only ones you can use to move horizontally, on the current line. Actually, long time Vim users rarely use them. Instead, we can use other motions in NORMAL mode to move faster, like these:

Keystroke	Description
`w`	Move forward to the beginning of the next word.
`W`	Move forward to the beginning of the next WORD.
`e`	Move forward to the end of the next word.
`E`	Move forward to the end of the next WORD.
`b`	Move backward to the beginning of the word.
`B`	Move backward to the beginning of the WORD.

A question arise: what's the difference between a "word" and a "WORD"? They represent two different motions. A "WORD" follows the usual concept of a word; a string of characters delimited by spaces.

You can think of a "word" as a keyword, containing only a specific set of characters. Mainly, it doesn't include some special characters.

For example, if you have the file "rank_01/functions.lua" from the book companion open, you can enter

```
-- A word and a WORD: see the difference?
local function restorePosition() {
```

Now, try to use the motions we've seen above to see the difference. Using "WORD" motions will skip the parenthesis, but the "word" motion won't.

Here are some more horizontal motions I find particularly useful:

Keystroke	Description
`f{character}`	To find a `{character}` after your cursor.
`F{character}`	To find a `{character}` before your cursor.
`t{character}`	Move till a `{character}` after your cursor.
`T{character}`	Move till a `{character}` before your cursor

After using one of these four keystrokes above, you can continue to move from character to character with:

Keystroke	Description
`;`	Move forward
`,`	Move backward

Help Yourself

`:help cursor-motions`

`:help left-right-motions`

Beginning and End of Line

Moving word by word can be slow and boring if you want to go to the beginning or the end of the line. Here are some more NORMAL mode keystrokes:

Keystroke	Description
`0`	Move to the first character of the current line.
`$`	Move to the last character of the current line.
`^`	Move to the first non-whitespace character on the current line.

A whitespace can be a space or a tab.

It's Playtime!

Exercise 1 - Horizontal Motions

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of `restorePosition`:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$")
    ↪   then
        vim.cmd([[normal! g`"]])
    end
end
```

How would you:

1. Move to the position first `f` of `function`?
2. Move back to the first `r` of `restorePosition()`?
3. Move to the end of the line?

Vertical Motions

Here are more NORMAL mode keystrokes, but to move your cursor vertically this time:

Motion	Description
`{line_number}G`	Move to the beginning of the line numbered `{line_number}`.

Motion	Description
	For example, `10G` will move the cursor on line 10.
`G`	Move to the last line of your file.
`1G` or `gg`	Move to the first line of your file.
`CTRL-u`	move **u**pward for half a screen.
`CTRL-d`	move **d**ownward for half a screen.
`CTRL-b`	move **b**ackward (upward) an entire screen.
`CTRL-f`	move **f**orward (downward) an entire screen.

You can also use COMMAND-LINE mode to move to a specific line number, with `:{line-number}`. For example, `:10` will move your cursor to line 10.

Finally, here are three last keystrokes allowing us to move to the top, middle, or last line of the actual display:

Motion	Description
`H`	Move to the top (**H**ome, or **H**ighest) of the window.
`L`	Move to the **l**ast line of the window.
`M`	Move to the **m**iddle line of the window.

It's Playtime!

Exercise 2 - Vertical Motions

In NORMAL mode, how would you:

1. Move to the 5th line of the file?
2. Move back to the very beginning of the file?
3. Move to the very end of the file?
4. Move multiple lines upward with one keystroke?

Help Yourself

`:help up-down-motions`

The Language of Vim (NORMAL Mode)

If you searched online some tutorials or other resources about Vim, I'm pretty sure you've seen this one: Vim has a language! You can speak with your editor!

In Vim, NORMAL mode keystrokes can be seen as "sentences", describing an action you want to perform. That's what I meant when I was saying that the keystrokes are *composable*. It's nothing less than brilliant.

These "sentences" are quite easy to understand. It allows us to link what we know already (the sentence) by what we need to learn (the NORMAL mode keystrokes).

Even better: knowing that Vim has a "keystroke language" will push you to combine them instinctively to do what you need to do, and, in many cases, it will work!

The Operators

We've learned how to walk in Vim with motions. It's time to perform some actions. To operate on our content.

The *operators* are the verbs of the Vim language. Here are three common operators:

Operator	Description
`d`	**d**elete
`c`	**c**hange
`y`	**y**ank (copy)

These operators won't do anything if you hit them in NORMAL mode. You need to combine them with motions. For example:

`d$`	To **d**elete from your cursor to the end of line. You can also use the alias `D`.
`dgg`	To **d**elete everything from the cursor to the beginning of the file.
`ggdG`	Move your cursor to the beginning of the file, and **d**elete everything till the end.

Let's explain a bit further what these three operator are doing:

- The **d**elete operator is self-explanatory.
- The **c**hange operator will delete and immediately switch to INSERT mode, effectively allowing us to... change our text.
- The **y**ank operator allows us to copy some of our text. Then, using the keystroke `p`, it can be **p**aste (the official term is **p**ut) somewhere else in your text.

By default, the keystroke `p` will paste the content after the character under the cursor. To put it before, use the uppercase variant of the keystroke `P`. The logic stays the same when you **y**ank whole lines and **p**ut them back.

I encourage you to try out all these operators. You can combine them with the motions we've seen in this chapter. Again, more practice you'll have, better you'll get! The exercises at the end of the chapter will help you to get there, too.

Help Yourself

`:help operator`

`:help objet-motions`

The Text-Objects

Instead of motions, we can also use another construct with our operators: the famous Vim text-objects. If the operators are the verbs of the Vim language, the text-objects are the nouns.

Simply put, a text-object is a set of character with a specific start and end. In Vim, “a word” is a text object, as well as “a sentence”, or “a paragraph”.

For example, you can use operators and text-objects in NORMAL mode as follows:

Keystroke	Description
`diw`	To **d**elete **i**nside the **w**ord. It deletes the current word under the cursor.
`daw`	To **d**elete **a**round the **w**ord. It deletes the current word under the cursor and its leading and trailing whitespaces.
`ciw`	To **c**hange **i**nside the **w**ord. It deletes the current word under the cursor and switch to INSERT mode. In short, you… change the word!
`dip`	To **d**elete **i**nside the **p**aragraph.

Note that each of these examples are composed of an operator and a text-object. For example, for the first example, `d` is the operator, `iw` is a text-object.

In Vim’s help, `daw` is described as **d**elete **a** **w**ord. I find this “translation” quite confusing, that’s why I use **a**round instead of **a**. Indeed, text-objects beginning with a “a” (like `aw`) often delete something more than the text-object beginning with “i” (like `iw`).

There are more text-objects you can use for your editing needs. You can discover new ones in the bonus exercises at the end of this chapter, or by looking at Vim’s help.

Help Yourself

```
:help text-objects
```

Undo and Redo (NORMAL mode)

I would be totally lost if I didn’t have any way to undo or redo my work. Here’s how to do so in NORMAL mode:

`u`	To **u**ndo your last edit.
`CTRL-r`	To **r**edo.

You can think of `CTRL-r` as you being in control (`CTRL`) of your content.

You’ll notice that whatever you’re doing in INSERT mode is equivalent to one undo. We’ll learn how to change this behavior later in the book.

It's Playtime!

Exercise 3 - Operators and Text Objects

Using the `hjkl` keys in NORMAL mode, move your cursor to the following position:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$")
     ↪  then
        vim.cmd([[normal! g`"]])
    end
end
```

1. How can you delete inside a word, keeping the space(s) surrounding it?
2. How can you undo what you just did?
3. How can you delete the word `local`, this time with the space following it?
4. Undo your change. How would you delete the word `local`, and directly switch to INSERT mode, only using 3 keys?
5. Again, undo your change. Move to the `a` of the word `local`, and hit `dw`. Can you explain what happened? What's the difference between `dw` and `diw`?

Help Yourself

`:help undo-redo`

Bending Vim to Your Will (Customization)

In Vim, almost everything is configurable. It's insane, I tell you. You can shape your editor according to your megalomaniac desires. Let's see the very basics here.

Creating Your Configuration

Your main configuration file should be in the following path by default, depending on what you use:

For Vim	`~/.vimrc`.
For Neovim	`$XDG_CONFIG_HOME/nvim/init.vim`

This file is sourced when Vim starts. Except that, they have nothing special; you can create another file with some Vimscript inside (including Ex commands) and source it manually if you want. We'll explore this functionality later in this book.

Whatever you're using, I'll call this configuration file *the vimrc* throughout the book.

As you can see, the path of Neovim's vimrc depends on the environment variable `$XDG_CONFIG_HOME`. It's most likely `~/.config`. If you don't know what are the XDG user directories, here's a good resource to learn more about them.

Let's now write our first lines of configuration. I would encourage you to write them using Vim. What you've learned in this first chapter should be enough for you to put your toes into Vim's relaxing waters.

To open this file, you can run `vim {path}` in your terminal. For example: `vim ~/.vimrc`.

First, let's add the following to our config:

```
noremap <Up> <Nop>
noremap <Down> <Nop>
noremap <Left> <Nop>
noremap <Right> <Nop>
```

The Ex command `noremap` allows us to bind a key to something else. We'll see this command more in details in the third chapter of the book.

Here, we assign the arrow keys to… nothing. This will force you to use `hjkl` instead. It might be a bit painful at first, but, trust me, it's for your own good. You can also conclude that I'm just a little pig, and then decide to use the arrow keys all your life; I'll be sad, but it's your right.

Here's another line we can add to our vimrc:

```
set clipboard+=unnamedplus
```

It will make the copy-paste mechanism less confusing, till you learn more about it in chapter IV.

We now have Ex commands on every line of our file. That's right: you could also run them in COMMAND-LINE mode, but they would be reset the next time you're closing Vim. In fact, you can add any Ex command in your vimrc, and Vim will automatically execute them during its startup, in order.

We've already seen that these commands have a long and short form. You can use either of them in COMMAND-LINE mode; but, when you write these Ex commands in a vimrc, I would encourage you to use the long form for a better readability.

If you use Vim instead of Neovim, we need to add these lines too:

```
" No compatibility with Vi
set nocompatible

" Display line numbers
set number

" Enhanced completion in command-line mode
set wildmenu

" Syntax highlighting
syntax on

" Enable filetype, indentation, plugin
filetype plugin indent on
```

As you can see, every line beginning with a double quote `"` is a comment.

Some explanations:

- `set nocompatible` - Vim looks less like Vi, its ancestor. That's great, because we don't want to use Vi, but Vim.
- `set number` - Display the line numbers.
- `syntax on` - Enable the syntax highlighting.
- `filetype plugin indent on` - Load a bunch of files to set automatically the indentation, and other options depending on the type of files open in Vim.

To see the effect of your new configuration, you can relaunch Vim and try to use your arrow keys. They don't work anymore. Great! Out of some constraints can come great creativity.

With all this knowledge in mind, I advise you to add some comments for everything you add to your vimrc, at least at the beginning. For example:

```
" Easier copy-paste from other applications to Vim
set clipboard+=unnamedplus
```

Help Yourself

`:help vimrc`

The Configuration Addiction

At that point, I'd like to warn you: configuring Vim can become addictive. Not I-lost-my-house-and-my-partner-left-me kind of addictive, but you can easily spend (too) many hours trying to come up with the best configuration in the universe.

My advice: just try to add what's useful for you, step by step. Don't try to recreate all the functionalities you had in your text editor or, even worst, your IDE. You'll get eventually there when we'll speak about plugin a bit later in this book but, before that, you should consider trying to understand and use the functionalities directly available in Vim.

Exercises

Basics

To solve these exercises, open with Vim the file `rank_01/functions.lua` from the book companion. You can do so by going into the root of the repository and run `vim rank_01/functions.lua`.

Don't forget that you can quit Vim with the Ex-command `:q`. Add a *bang* to the command to quit without saving: `:q!`.

Each exercise has a series of question. You should solve them in order. The changes done for each question is the starting point for the next one.

Exercise 4 - Horizontal Motions, the Return

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of the following line:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

Using only one key per question, how would you:

1. Move to this position: `    if vim.fn.line("'\"") > 1`?
2. Move to this position: `    if vim.fn.line("'\"") > 1`?
3. Move to this position: `    if vim.fn.line("'\"") > 1`?
4. Move to this position: `    if vim.fn.line("'\"")`?

Exercise 5 - Operators, Motions, and Text-Objects

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of the word "vim" as follows:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

1. How would you move to the first parenthesis of the same line, only using two keys?
2. How would you delete everything inside these parentheses?
3. Undo your change.
4. How would you move your cursor at the "i" of "line", using only two keys (including the key `i`), as follows: `if vim.fn.line("'\"") > 1`?
5. How would you move to the "i" of "vim", using only one key, as follows: `if vim.fn.line("'\"") > 1`?

6. While staying on the same line, try to use `fn`, `Fn`, `tn`, `Tn`.Then, try to use `;` or `,`, to get used to these movements. Try to replace "n" with other characters present on the line, too.

Beyond the Basics

These exercises dive deeper in some concept seen in the chapter.

Exercise 6 - More Text Objects!

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of the function "restorePosition" as follows:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

A paragraph is also a text object. It starts on an empty line, and end on the next empty line.

1. How would you delete the whole function `restorePosition` only using three keys?
2. Undo your change. How would you delete the next block of parenthesis, only using three letter keys?
3. Undo your change. How would you delete a sentence?
4. To find the start and end of the text-object "sentence", what Ex-command would you use?
5. What text-object could be useful to edit some HTML? Do you think this text-object exists in vanilla Vim?

Exercise 7 - More Motions!

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of the function name "restorePosition" as follows:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

1. We've seen the text-object "paragraph" in the previous exercise. Using Vim's help, try to find what motion allowing you to move from one paragraph to another.
2. How would you delete the letter `r` in `local function restorePosition()` with two letter keys? One?
3. Undo your change. How would you delete the space before your cursor: `local function restorePosition()` with two letter keys?
4. The answers of the two previous questions use motions or text-objects?

Exercise 8 – Up and Down Following Indentations

Using the `hjkl` keys in NORMAL mode, move your cursor on the character `l`, at the beginning of the word `local`:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

How would you:

1. Move to the next line, on the `i` of `if`, only using one key?
2. Move back to the starting position, only using one key?

Solutions

Exercise 1 – Horizontal Motions

Question	Keystroke	Result
start		`local function restorePosition()`
1.	`b`	`local function restorePosition()`
2.	`w`	`local function restorePosition()`
3.	`$`	`local function restorePosition()`

Exercise 2 – Vertical Motions

Question	Keystroke	Result
start		`#!/usr/bin/env lua`
1.	`5G`	``        vim.cmd([[normal! g`"]])``
2.	`1G` or `GG`	`#!/usr/bin/env lua`
3.	`G`	`}`
4.	`CTRL-u`	``        vim.cmd([[normal! g`"]])``

Note that you can also move at the fifth line of the file in COMMAND-LINE mode, with `:5`.

Exercise 3 – Operators and Text Objects

Question	Keystroke	Result
start		`local function restorePosition()`
1.	`diw`	` function restorePosition()`
2.	`u`	`local function restorePosition()`
3.	`daw`	`function restorePosition()`
4.	`ciw`	`| function restorePosition()`
5.	`dw`	`locfunction restorePosition()`

5. All the NORMAL mode keystrokes `dw` , `daw` , and `diw` use the operator `d` (for `d` elete).

The difference between these keystrokes: `w` is a motion, `aw` and `iw` are text-objects.

If you:

- Use a motion without operator, you'll move your cursor.
- Use a motion with an operator, you'll operate from the cursor position to the destination of the motion.

That's why `dw` , in this example, delete from the cursor position to the beginning of the next word `function` .

Said differently, the start and the end of a text-object are always the same: for example, the start of `aw` will be the first letter of the word the cursor is on, and the end will be the potential space after the word.

Another difference: a text-object always need to be prefixed by an operator. A motion can be used on its own.

Exercise 4

Question	Keystroke	Result
start		`    if vim.fn.line("'\"") > 1`
1.	`^` or `w`	`    if vim.fn.line("'\"") > 1`
2.	`w`	`if vim.fn.line("'\"") > 1`
3.	`W`	`if vim.fn.line("'\"") > 1`
4.	`0`	`    if vim.fn.line("'\"")`

Exercise 5

Question	Keystroke	Result
start		`if vim.fn.line("'\"") > 1`

Question	Keystroke	Result
1.	`f(` or `t"`	`if vim.fn.line("'\"") > 1`
2.	`da(` or `dab`	`if vim.fn.line > 1`
3.	`u`	`if vim.fn.line("'\"") > 1`
4.	`Fi`	`if vim.fn.line("'\"") > 1`
5.	`;`	`if vim.fn.line("'\"") > 1`

Exercise 6

Question	Keystroke	Result
start		`local function restorePosition()`
1.	`dap` or `dip`	`local function deleteTrailingWS()`
2.	`dab`	`local function restorePosition`
3.	`das` or `dis`	``g`"]])``

4. You first need to go to COMMAND-LINE mode by typing `:`. Then, you need to use the help Ex-command `help`, followed by what you want to find, i.e `sentence`, or `text-object`. In short: `:help sentence` or `:help text-boject`.
5. The text object `at` or `it` stand for "around an HTML tag" and "inside an HTML tag", respectively. Add whatever operator as a prefix to edit your HTML easily.

Exercise 7

1. You'll find this info by running the Ex-command `:help object-motions`. You can use the motion `{` to move up a paragraph, `}` to move down.

Other than that, you could try `:help cursor-motions` and search for "paragraph", by running `/paragraph`.

Question	Keystroke	Result
start		`local function restorePosition()`
2.	`dl` or `x`	`local function estorePosition()`
3.	`dh`	`local functionrestorePosition()`

4. We're using motions here: `l` is the motion to move one character to the right, `h` to move one character to the left.

The letter `l` is also a useful mnemonic for `l`etter. Even if it's easier to use `x` to delete a character, you can use other operators with the motion `l`, like `yl` if you want to copy it.

Exercise 8

Question	Keystroke	Result
start		`local function restorePosition()`
1.	`+`	`if vim.fn.line("'\"") > 1`
2.	`-`	`local function restorePosition()`

Rank II – Vim for Beginners

Sitting down on the beach of this unknown territory, your feelings of being lost dissipate a bit. Maybe understanding some basics about the Vim editor helped? You need to find some sort of shelter however; even if it's still warm and sunny outside, the night will come.

The beach is in fact quite small; there are high rocks all around you, sharp like razors. You have no choice: you need to adventure in the jungle in front of you.

While you're waling toward it, you see a sign on a high palm tree:

"Welcome! Shelter in 50 meters".

You enter the dense jungle. After a couple of meters only, you notice a small wooden house. It looks old but solid. A lot of vegetation is growing around it, and there is even small tree on the low roof.

You pause a moment. There's something odd in this jungle: you don't hear anything. No birds, no other animals. Nothing. Only the quiet, regular sound of the waves from the beach behind you.

Even if the vegetation is wild and very dense, you manage to find a way to enter the small house. Inside, there is no vegetation at all; as if no plants were allowed to enter. There is only three things in there: a chair, a little table, and a bed.

After entering, the entrance door suddenly close behind you. Worried, you try to open it. It's blocked. You try harder, without success. No doubts: there's something blocking the door.

Trying to keep your calm, you look around. There is no other way to go outside! Not a window, a whole, or anything else. Who designed this place? Cavemen?

Even if the place has no bulb or candle, there is a diffuse light all over the room.

Your sight goes on the table. There is a basket of apple as well as a strange object on it, square and thin. It reminds you of the "save" icons in many computer programs on your high-end computer at home. You believe this is a floppy disk; sometimes, your grandmother talk about it in her violent nostalgia crises.

You put the computer on the table. You notice that it seems quite old too, and it seems that it's equipped with a reader able to take the floppy disk. You insert it. Suddenly, Vim, still running and displayed on the screen, open it:

"Welcome to your new house! At least for the night, and maybe a couple more. Maybe all your life. Who knows?

You see, you can't go out anymore. You're trapped here. I'm sorry, but it's necessary.

The basket of apple will automatically grow some new apples if you eat them all. I hope you like apples! What kind of monster wouldn't like them, anyway? I know you're not this kind. You'll like apples. Nothing is better!

You can only go out if you do all the exercises of this chapter. You need to get better at Vim, you know. Otherwise, you're doomed to have a strict apple diet. Personally, I'd be find with that. Did I tell you that I lived in this room for a couple of centuries? Only playing with Vim. Ah! Good old times.

Oh, I almost forgot to tell you: **don't panic!**

Anyway. Here's the summary of this chapter:

- We'll dive more into the NORMAL and INSERT modes, and we'll introduce the VISUAL mode.
- We'll see more ways to delete some text, and the curious side-effects doing so.
- I know you'll panic. They all do. That's why we'll see more in depth how to use Vim's internal help. It's great, apple-like awesome.
- Finally, we'll continue to build Vim's personal configuration.

To try and experiment with what we'll see throughout this chapter, I advise you to open the file `rank_01/functions.lua` from the book companion.

Last thing: I've put fresh sheet in your bed. If you're too tired to continue playing with Vim, just rest a bit and take a couple of apples. This floppy disk will still be here tomorrow!

The Modes of Vim, the Return

Let's continue exploring the different Vim's Modes, by digging deeper into the NORMAL and INSERT mode, as well as discovering the VISUAL mode.

Counting Motions or Text-Objects in NORMAL mode

Throughout this book, we'll see that Vim is the champion for automating mundane, annoying tasks we need to do too often. What doesn't require our human brain should be automated, to use what's left of our neurones to more important thoughts and tasks.

Most NORMAL mode motions we've seen in the previous chapter can be repeated as many times as we want. To do so, we need to add add a *count* before the motion. The count determines how many times the motion will be applied.

Actually, there's always a count applied to a motion when you used them; it just happen that the default, in most case, is 1. It's also true for many NORMAL mode keystrokes.

For example, you can open the file "rank_01/functions.lua" from the book companion in Vim, and you can try to put your cursor on the first character of the 3rd line, as follows:

```
local function restorePosition()
```

Now, try to hit `2w` in NORMAL mode. Here's the result:

```
local function restorePosition()
```

You've applied the motion "word" two times! You can try to add a count to the motions you already know to get used to it.

If you apply a count to both a motion and an operator, they are multiplied. For example, if you hit `2d3w`, you'll delete 6 words.

Now you might wonder: is it really useful? Do you see yourself counting words or other text-objects, to move where you want? If you experiment a bit with counts, you'll see that it's not easy to look ahead and know exactly what count you need.

That said, when you don't want to move your cursor too far away, count can be useful. Not so much when you need to do an operation multiple lines away.

We'll come back to this functionality throughout the book, to see more practical applications of counts.

From NORMAL to INSERT Mode

As we saw in the first chapter, Vim requires you to switch between NORMAL and INSERT mode quite often. That's why there are many NORMAL mode keystrokes to switch to INSERT mode in slightly different ways.

Here are the main ones:

Keystroke	Description
`i`	To insert content before the current character.
`a`	To insert content after the current character.
`A`	To insert content after everything, at the end of the line.
`o`	To open a new line below the current one.
`O`	To open a new line above the current one.
`ESC` or `CTRL-c` or `CTRL-[`	Switch back from INSERT mode to NORMAL mode.

The more you'll use these keystrokes, the more seemliness your switch between NORMAL and INSERT mode will be. Soon, you won't even notice that you switch between both. Guaranteed!

Help Yourself

`:help Insert-mode`

The VISUAL Mode

There is a third important mode in Vim you'll often use: VISUAL mode. Its goal? Selecting a piece of text.

How to switch from NORMAL mode to VISUAL mode? You might already have guessed it: you need to hit `v` in NORMAL mode. You'll see the indicator `--VISUAL--` appearing in the bottom left corner of Vim. The selection will start at the cursor position. You can hit some motions to extend the selection, and then use an operator to operate on the selected text.

When hitting `v` in NORMAL mode, you enter the VISUAL mode per character. If you want to select entire lines at once, you can enter VISUAL mode linewise. To do so, you need to use `SHIFT-v` (or V) (in uppercase), and then going up and down will select what you want.

To summarize:

Keystroke	Description
`v`	Switch to VISUAL mode per character.
`V`	Switch to VISUAL mode linewise.

There is also the VISUAL mode blockwise, but we'll see that later in the book.

A last thing: VISUAL mode is convenient because it allows us to operate upon a specific portion of text, in a very... visual way. That said, it's often quicker for Vim experts to only use motions, text-objects, and count, because you don't spend your time selecting text. So, if you want an advice, try to train yourself not using VISUAL mode. Not necessarily at the beginning, but when you'll be more comfortable with Vim.

As usual, to come back from VISUAL mode to NORMAL mode, press `ESC`.

Help Yourself

`:help visual-mode`

The REPLACE Mode

The last mode I'd like to highlight is the REPLACE mode. As it names indicate, it's a mode where you can replace some text.

To replace a first character by a second one:

1. Move your cursor on the first character.
2. Hit `r` in NORMAL mode.
3. Hit the second character.

And voila! As usual, you can also use a count to replace a specific number of characters with one character.

If you want to replace more than one character, you can hit `R` in NORMAL mode. You'll then replace everything you're typing.

To summarize:

Keystroke	Description
`R`	Switch to replace mode.
[count] `r`	Switch to replace mode for `[count]` characters (default: 1).

You already know how to come back from REPLACE mode to NORMAL mode: yes, with the `ESC` key!

It’s Playtime!

Exercise 1 - Vim’s Modes

Using the `hjkl` keys in NORMAL mode, move your cursor at the following position:

```
local function restorePosition()
        vim.cmd([[normal! g`"]])
    end
end
```

For each question, undo all your changes and come back to the initial cursor position.

How would you:

1. Move to the third `v` on the line, using only a NORMAL mode keystroke of 3 characters?
2. Replace the first `vim` of the line with `nop` using REPLACE mode?
3. Delete the whole line using VISUAL mode?
4. Change the first `v` of `vim` by a `w`, by using a NORMAL mode keystroke of 1 character followed by one letter in INSERT mode?
5. Create a new line below and enter INSERT mode, using only a NORMAL mode keystroke of 1 character?
6. Insert a semi colon “:” at the end of the line, by using a NORMAL mode keystroke of 1 character followed by the semi-colon in INSERT mode?

Help Yourself

`:help replace-mode`

Deleting In Vim (NORMAL mode)

In the first chapter, we’ve already seen how to delete text with the operator `d`. With a motion or a text-object, we’ve already have a powerful tool to get rid of all these annoying character.

Cross What You Don’t Want

Here are two useful keystrokes to delete single characters in NORMAL mode:

Keystroke	Description
`x`	Delete the character under the cursor
`X`	Delete the character before the cursor

Both are aliases for `dl` and `dh`, which are the combinations of the delete operator and the motions "l" (right) and "h" (left).

Deleted Content To

There is something many Vim beginners find surprising, and quite annoying. For example, let's say that you've the following content:

```
local function restorePosition()
```

Then, let's try the following in NORMAL mode:

1. Hit `y` to **y**ank (copy) the word `local`.
2. Then, move to the `f` of `function` by hitting the motion `w`.
3. Let's then hit `diw` for **d**elete **i**nside **w**ord.
4. Let's now hit `p` to **p**aste (or **p**ut) our content. What will be inserted here?

Here's the result

```
local  functionrestorePosition()
```

If you did all the steps above, you'll see that `function` will be added to the line, even if you copied `local` before!

For now, you could think of this behavior as throwing whatever you delete into your clipboard, as it was a trash bin. This will make more sense when we'll look at Vim's registers, a great functionality. From there, we'll have the opportunity to choose what we want to paste (what we've yanked, what we've deleted, and more). Till then, bear with me; it might be quite annoying at first, but it's worst the pain.

It's Playtime!

Exercise 2 - Deleting in Vim

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of `restorePosition`:

```
local function restorePosition()
    if vim.fn.then
        vim.cmd([[normal! g`"]])
    end
end
```

Use Vim's search for each question, from the cursor position you moved on for the previous question. How would you:

1. Move to the `r` of `restorePosition`?
2. Delete the word `restore` using a three letter keystroke in NORMAL mode?
3. Replace the uppercase `P` of position with a lowercase `p`?
4. Move to the `n` of `position`?
5. Paste what you've deleted, to end up with `positionrestore`?
6. Delete the `e` of `positionrestore` with one-letter NORMAL mode keystroke?

Help Yourself

`:help deleting`

Searching (COMMAND-LINE Mode)

We can hit `:` in NORMAL mode to switch to COMMAND-LINE mode and run Ex commands, as we saw in the first chapter. We can do much more in COMMAND-LINE mode, however.

If you hit `/` or `?` in NORMAL mode, you'll also switch to COMMAND-LINE mode. But this time, it's not for running Ex commands; it's for searching a pattern in our text.

Here's the basics:

Command	Description
`/{pattern}`	Search `{pattern}` forward from the cursor position.
`?{pattern}`	Search `{pattern}` backward from the cursor position.

When you hit `ENTER` after typing your `{pattern}`, you'll switch back to NORMAL mode and you'll move directly to the first match in your text. If there is no match, Vim will display the error "Pattern not found".

If there are more than one match, you can move to them by repeating the search. Here are the different NORMAL mode keystrokes to do so:

Keystroke	Description
`n`	Repeat the last search forward (move to the next match).
`N`	Repeat the last search backward.

That's not all: searching is a motion. It means that you can use an operator (as we saw in the first chapter), and then type your search. The operator will operate from the cursor position to the first match.

Let's take an example in our favorite file:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

If you then hit `/` in NORMAL mode, you'll move to the bottom of the screen. From there, you can type the pattern you want to search for. For example: `vim`. Hitting `ENTER` will bring your cursor to the next string "vim" and switch back to NORMAL mode.

Then, hitting `n` two times will bring you at the end of the line:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

It's Playtime!

Exercise 3 - Vim Search

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of `restorePosition`:

```
#!/usr/bin/env lua

local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$")
    ↪  then
        vim.cmd([[normal! g`"]])
    end
end
```

Using Vim's search, how would you:

1. Move your cursor to this position: `    if vim.fn.line("'\"") > 1`
2. Move to the next match of your search pattern?
3. Move back to the first match of your search pattern?
4. Move to the 4th match of your search pattern, using a count?

Navigating Vim's Help (COMMAND-LINE Mode)

As we saw in the first chapter, Vim's help is your most previous ally in your Vim's adventure. I don't know everything regarding Vim, and this book won't even explain everything I know. Many details are left out, for this book not being too boring, and for you not to be lost in details.

In contrast, vim's help knows almost everything. That's why, if you need to dig deeper into a functionality, it should be your first reflex. Stack overflow won't cut it.

Basics

Here are the most useful help commands you can use:

Ex Command	Short Name	Description
`:help`	`:h`	Open the main help file.
`:help {subject}`	`:h {subject}`	Open the help about `{subject}` in a split window.
`:helpgrep {pattern}`	`:helpg {pattern}`	Search the `{pattern}` in the help files.

I always reach for `:help {subject}` first when I need to look at a specific functionality. If I don't find what I want, I reach then for `:helpgrep {pattern}`; this Ex command will have a bigger chance to find a false positive, however.

Help Yourself

`:help helphelp`

`:help helpgrep`

Follow The Definition

Vim's help are simple text files. Thanks to a tag file, you can also jump to the definition of some specific words. We'll come back to this concept of tags much later in this book.

For now, let's just say that you can get to the definition of some keywords in this set of help files. This is really useful to look at related subjects you might want to dive into. If you're syntax highlighting is on (running `:syntax on` in COMMAND-LINE mode, as we saw in the first chapter), these keywords will appear in different colors in Vim. If not, you can find them between two pipes `|`.

If you want to follow one of these keywords, you can place your cursor on it and hit `CTRL-]`. You'll then *jump* to another part of the same file, or even to another file. To jump back, you can hit `CTRL-o`.

Let's try it. First, run the Ex command `:help`. A new window will open vertically. Welcome to the main help file!

Then, place your cursor on `bars`, as follow:

```
Close this window:  Use ":q<Enter>".
   Get out of Vim:  Use ":qa!<Enter>" (careful, all changes are lost!).

Jump to a subject:  Position the cursor on a tag (e.g. |bars|) and hit CTRL-].
   With the mouse:  Double-click the left mouse button on a tag, e.g. |bars|.
    Jump back:  Type CTRL-O.  Repeat to go further back.
```

Finally, hit `CTRL-]`. You'll jump to the end of the same file. To come back to where you were, hit `CTRL-o`. To close the window, you can run `:q` in COMMAND-LINE mode.

This is a very effective, and useful way to navigate Vim's help files!

A last tip: to display the table of content, you can hit `gO` in NORMAL mode. It will open a new window where you can select the subsection you want to read. If the window is already open, hitting `gO` will move your cursor inside it.

Again, to close the window, you can run `:q` in COMMAND-LINE mode.

To summarize:

Keystroke	Description
`CTRL-]`	Jump to the definition of the keyword under the cursor.
`CTRL-o`	Jump back to your older cursor position.

Help Yourself

`:help`

`:help CTRL-]`

`:help CTRL-O`

Finding What Your Heart Desire

We've seen just above that you can search for different functionalities using `help {subject}`. But what should be this `{subject}`?

You can try to find it by yourself of course, but it's sometimes difficult to really know where to find the information you seek, especially when you get familiar with Vim. That's why this book gives you many help Ex commands for each section.

Here are other, more precise patterns you can use depending on what you want to find:

Ex Command	Description
`:help CTRL-{char}`	Open the help for a keystroke with `CTRL` and a `{char}` in NORMAL mode.
`:help i_CTRL-{char}`	Open the help for a keystroke with `CTRL` and a `{char}` in INSERT mode.
`:help c_CTRL-{char}`	Open the help for a keystroke with `CTRL` and a `{char}` in COMMAND-LINE mode.
`:help -{arg}`	Open the help for the `{arg}` we can pass to the "vim" command.

To illustrate, here are some examples of Ex commands you can run in Vim:

Ex Command	Description
`:help CTRL-o`	Help for the NORMAL mode keystroke `CTRL-o`.
`:help i_CTRL-o`	Help for the INSERT mode keystroke `CTRL-o`.
`:help c_CTRL-f`	Help for the COMMAND-LINE mode keystroke `CTRL-f`.
`:help -u`	Help for the Vim's command line option `-u`.

There are many more pattern for help's subjects. We'll discover them as we introduce the concepts they are about.

Vimscript or Lua? (Customization)

The content of your vimrc and the different plugins for Vim are Ex-commands, as we saw. But Ex-commands are just a part of larger programming language called Vimscript.

At the beginning of time, Vimscript was created to configure Vim, and nothing more. It's straightforward to use for this limited job; Vimscript has useful constructs to customize Vim as we see fit. That's why we use it.

But, over the years, more and more constructs were added to Vimscript. Step by step, it became a general programming language. It's where things become to get ugly. As a programming language, Vimscript has many pitfalls and weird design decisions. It's painful to understand, use, and debug.

Vimscript is not the only option, nowadays. It's also possible to use Lua to configure Vim. It's where the difference between Vim and Neovim is the most obvious: while Neovim allow you to configure almost everything in Lua, Vim doesn't allow you as much.

The next versions of Vim will come with Vimscript 9, a new and faster version of the language which is trying to fix Vimscript quirks. But, as I write these lines, it's not officially out yet.

All in all, most of the configuration of this book will be in Vimscript. When we'll need a more general programming language to push the customization further, I'll give both Vimscript and Lua versions.

Why not doing everything in Lua at the first place, you might wonder? For different reasons:

1. Vim users can't configure as much in Lua as Neovim's users.
2. Many good resources and useful functions available out there are written in Vimscript. Knowing a bit about it can help you understand them, and even modify them for your needs.
3. Vimscript will be likely supported for a long time by both Vim and Neovim.
4. Many Vimscript functions can be called from Lua script. Knowing them is essential, even if you prefer doing everything in Lua.

If you really want to write your whole configuration in Lua, you can find 43270724983 resources online to do exactly that. I'm not kidding; it's trendy nowadays to switch your vimrc from Vimscript to Lua.

Vim's Options (Customization)

We've began to configure our vimrc in the next chapter. Let's now look at a cornerstone of Vim configuration, the famous Vim's options.

Defining Options

You can modify Vim's general behavior by modifying its options. You can think of options as variables with some scope; you can display their values, or modifying them using Ex commands in COMMAND-LINE mode. An option can be either a boolean, a string, or a number.

If you want to permanently modify Vim's options, you need to assign their new values directly in your vimrc, as we did with the option `clipboard` in the previous article for example.

Here are the commands you can use: to manage these options:

Ex Command	Description	Type
`:set no<option>`	Unset the `{option}`	Boolean
`:set <option>!`	Toggle the `{option}`	Boolean
`:set <option>?`	Return the `{option}`'s value	All
`:set <option>&`	Reset the `{option}` to its default value	All
`:set {option}={value}`	Set the `{value}` to the `{option}`	String or number
`:set {option}+={value}`	Add the value `{value}`	Number
	Append a string `{value}`	String
`:set {option}-={value}`	Subtract the value `{value}`	Number
	Delete the string `<value>`	String

For example, if you want to display the filetype of the current file open, you can run this Ex command:

```
:set backspace?
```

The string option `backspace` is made of multiple substrings separated with commas ','. You can delete one of these substrings as follow:

```
:set backspace -= indent
```

You can now look at its new value:

```
:set backspace?
```

To come back to its default value, you can run:

```
:set backspace&
```

Remember: you don't need the prefix `:` if you want to put some Ex commands in your vimrc, like setting some options.

A last important thing: if you want to get some information about options in Vim's help, you need to surround your pattern with single quotes. For example:

```
:help 'number'
:help 'swapfile'
```

Help Yourself

`:help options`

`:help set-options`

`:help option-list`

Setting Options: An Interactive Way

There's another way to modify Vim's options: using the Ex command `:options`.

If you run it, another window will open. From there, you can see all the options you have access to, and you can set them by hitting ENTER on the value you want. For example:

```
compatible  behave very Vi compatible (not advisable)
    set nocp   cp
```

The cursor here is on `nocp`, so if you hit `ENTER` you'll set the option `compatible` to false. For string values, you can modify them and then hit `ENTER` to set them.

These options are grouped by general functionality, so it can be a good way to experiment and discover new options. The number of options can be a bit daunting, however; don't try to set all of them at once, or you'll burn out by so many… possibilities!

Useful Options

Here are the options we've seen in the previous chapter:

Option	Description	Type	Recommendation
`compatible`	Vim becomes compatible with Vi	Boolean	False
`number`	Display line numbers	Boolean	True
`wildmenu`	Enhanced completion in COMMAND-LINE mode.	Boolean	True

Here are three more interesting options related to the search we've seen above:

Option	Description	Type	Recommendation
`'ignorecase'`	Search is not case sensitive	Boolean	true
`'smartcase'`	Search is case sensitive if the pattern has one more uppercase. Need `ignorecase` to be true.	Boolean	true
`'showcmd'`	Show partial NORMAL mode keystrokes on the right of the command-line window		Boolean true
`'hlsearch'`	Highlight the matching search pattern. Use `:nohlsearch` (or `:noh`) to turn the highlight off. The next matching pattern will be highlighted, however.	Boolean	true

We'll see many more options throughout the book you'll have the occasion to set up for your own specific needs.

It's Playtime!

Exercise 4 - Manipulating Options

How would you:

1. Get the value of the option 'filetype'?
2. Toggle the value of the option 'number'?
3. Switch the value of the option 'compatible' to false?
4. Append the substring "S" to the existing value of the option 'shortmess'?
5. Only delete the substring "S" to the existing value of the string option 'shortmess'?
6. Set back the option 'shortmess' to the default value?

Options are useful, and we'll use them all the time in the following chapters. Be prepared to fill your life with them!

Exercises

Basics

Exercise 5 - Help Yourself!

Using Vim's help, how would you:

1. Find information about the REPLACE mode?
2. Find information about the keystroke `*` in NORMAL mode?
3. Find information about the keystroke `CTRL-a` in INSERT mode?
4. Find information about the keystroke `CTRL-f` in COMMAND-LINE mode?
5. Find information about the option `iskeyword`?

Exercise 6 - Search Highlighting

Using what you've seen in this chapter and Vim's help, how would you:

1. Set the option 'hlsearch'?
2. Search for the word `vim`.
3. The matching pattern `vim` should now be highlighted. How would you turn off the highlight, but still highlight the next search you'll do?
4. How can you totally disable matching pattern highlight?

Exercise 7 - More Vim Search

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of `restorePosition`:

```
local function restorePosition()
    if vim.fn.then
        vim.cmd([[normal! g`"]])
    end
end
```

Use Vim's search for each question, from the cursor position you moved on for the previous question. How would you:

1. Move your cursor to this position: `vim.fn.line("$") then`
2. Move your cursor to this position: `vim.cmd([[normal! g`"]])`
3. Move your cursor to this position: `if vim.fn.line("'\"") > 1`
4. Yank everything until the word `then` at the end of the line.
5. Delete everything until the word `then` at the end of the line.

Beyond the Basics

Exercise 8 - Swapping

Using the `hjkl` keys in NORMAL mode, move your cursor on the character `l` at the beginning of the word `local`:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

Undo all your changes after each question to come back to the starting position above.

How would you:

1. Swap the character `l` you're on, with the following character `o`, only using two keys?
2. Swap the current line you're on, with the line just below, only using three keys?
3. Swap the word `local` with the word `function` using five keys?

Solutions

Other solutions than the one presented here are possible.

Exercise 1 - Vim's Modes

Question	Keystroke	Result
start		`    if vim.fn.line("'\"")`
1.	`2fv`	`<= vim.fn.line("$") then`
2.	`Rnop`	`    if nop.fn.line("'\"")`
3.	`Vd`	``     vim.cmd([[normal! g`"]])``
4.	`rw`	`    if wim.fn.line("'\"")`
5.	`o`	
6.	`A:`	`<= vim.fn.line("$") then:`

Exercise 2 - Deleting in Vim

Question	Keystroke	Result
start		`local function restorePosition()`
1.	`fr`	`local function restorePosition()`
2.	`dtP`	`local function Position()`

Question	Keystroke	Result
3.	`rp`	`local function position()`
4.	`fn`	`local function position()`
5.	`p`	`local function positionrestore()`
6.	`x`	`local function positionrestor()`

Exercise 3 – Vim's Search

Question	Keystroke	Result
start		`#!/usr/bin/env lua`
1.	`/vim` then `ENTER`	`if vim.fn.line("'\"") > 1 and vim.fn.line("'\"")`
2.	`n`	`if vim.fn.line("'\"") > 1 and vim.fn.line("'\"")`
3.	`N`	`if vim.fn.line("'\"") > 1 and vim.fn.line("'\"")`
4.	`3n`	`vim.cmd([[normal! g`"]])`

Exercise 4 – Manipulating Options

1. `:set filetype?` or `:set ft?`
2. `:set number!` or `:set nu!`
3. `:set nocompatible` or `:set nocp`
4. `:set shortmess+=S` or `:set shm+=S`
5. `:set shortmess-=S` or `:set shm-=S`
6. `:set shortmess&` or `:set shm&`

Exercise 5 – Help Yourself

1. `:help replace-mode`
2. `:help *`
3. `:help i_CTRL-a`
4. `:help c_CTRL-f`
5. `:help 'iskeyword'`

Exercise 6 – Search Highlighting

1. `:set hlsearch`
2. `/vim`
3. `:nohlsearch` or `:nohl`
4. `:set nohlsearch` or `:set hlsearch!`

Exercise 7 – More Vim Search

Question	Keystroke	Result
start		`local function restorePosition()`
1.	`/line` then `nn`	`vim.fn.line("$") then`
2.	`/vim`	`vim.cmd([[normal! g`"]])`
3.	`?line`	`    if vim.fn.line("'\"") > 1`
4.	`y/then`	`    if vim.fn.line("'\"") > 1`
5.	`d/then`	`    if vim.fn.then`

Exercise 8 – Swapping

Question	Keystroke	Result
start		`local function restorePosition()`
1.	`xp`	`olcal function restorePosition()`
2.	`ddp`	`local function restorePosition()`
3.	`dwelp`	`function local restorePosition()`