Learning to play Vim

Master the best text editor,
from Beginner to Composer

Matthieu Cneude

Learning to Play Vim

Matthieu Cneude

Learning to Play Vim

Contents

Preface

It has been said, time and time again, that using Vim (or Neovim) is a challenge only the most powerful wizards of the sword coast can tackle.

I disagree. Vim can be used by anybody. You just need to put what you know about text editors on the side for a little while. Vim works differently indeed; and that's why it's powerful.

It's easy to learn the basics of Vim. In fact, the first chapters of this book are enough for you to edit any text file.

But it's also true that Vim takes time to master. You can improve your workflow each time you use Vim if you want to. It's a benefit, not a drawback. To me, Vim is the gamification of writing: I can focus on my writing 90% of the time, and I'll spend the last 10% messing around, trying to make my editing power even greater while having fun.

This is what this book is about: we'll go on the quest to discover (or re-discover) the Best Text Editor in the World™. And, more importantly, trying to have fun too!

This book can be read by any Vim enthusiast out there, from the perfect beginners who want to write their autobiographies in 25 tomes using Vim, to the developers using Vim for 25 years and still trying to improve their workflows. Whoever you are (a fine human being I'm sure!), I'm confident you'll learn something new here.

Now, if you're an innocent beginner, a question will quickly arise: should you use Vim or Neovim?

Vim Or Neovim? That's the Question

Vim is itself the product of different text editors which came before it. At one point in time, some developers decided that Vim needed to take another direction moving forward. They basically took Vim source code and created Neovim.

As a result, both Vim and Neovim are very similar. At the same time, they have noticeable differences too.

I'll use the name "Vim" throughout this book to speak about both Vim and Neovim. That said, if one option or command works for one but not the other, I'll always specify it in this book.

We can expect even more differences between Vim and Neovim as time passes. For now, here are the main ones:

1. Vim uses the scripting language Vimscript for its configuration. With Neovim, you can use Vimscript or Lua; Lua is a simpler (and more consistent) scripting language to configure everything.

2. Neovim has some sane defaults, while Vim needs to be configured a bit more extensively. On top, you might need to compile Vim with the features you want, while Neovim include most things you'll need by default.
3. There are many more tools available in native Neovim for software developers: it supports LSP (Language Server Protocol) out of the box for example, or treesitter to enable a more powerful syntax highlighting.

Also, Neovim is evolving faster than Vim, which is a good and a bad thing. It's good because new functionalities are often added, but it's also bad because your configuration might get outdated, and, in the worst case, break some functionalities. You basically need to debug your configuration in that case.

All in all, I would recommend you to use Neovim, only to be able to use some Lua for more advanced configuration. That said, if you want to write prose instead of writing code, Vim will be more stable without major disadvantages.

In this book, I provide both Vimscript and Lua for everything we'll configure. If you stick to Vimscript, you can switch between Vim and Neovim if you change your mind later in your journey. In practice, there are little advantages to write all your configuration in Lua; this language is only useful more advanced customization.

We'll discuss all of that in more details in the book.

In a nutshell: don't stress it. Pick Vim or Neovim, you'll be able to transfer what you'll learn from this book from one editor to the other.

What This Book is not About

First and foremost: this book is only about Vim (or Neovim) running in a Unix-like shell, in a terminal. I strongly believe that Vim is most powerful in this context. As a result, I won't speak about Vim with Graphical User Interfaces (GUIs).

You're also at the wrong address if you want to learn Vimscript 9 (the new Vimscript introduced in Vim 9, but not available in Neovim): I have no clue about it, and I won't cover it here.

I won't describe the shell and its functionalities in this book. That said, if you're interested to learn more about the shell, and, more generally, about the tools you can use to create a Mouseless Development Environment, it's the best moment for me to plug in there my other book. It's my life mission to spread the love of the keyboard. I'm only getting started here.

Vim and Neovim can be extended with external plugins. While I'll give you my personal recommendations, this book is not about plugins, but more about Vim's vanilla functionalities.

Finally, this book is not about re-creating your favorite IDE in Vim. Your IDE is your IDE, and it will always be different, even if you install an unhealthy amount of plugins. That said, coupled with the power of the shell, you can definitely make Vim (especially Neovim) looking a lot like a streamlined IDE.

What You Need to Follow Along

As I was writing above, this book is specifically about Vim running in a terminal: you'll need a shell like GNU Bash or Zsh. If you're using macOS or any Linux distribution, you already have a

shell. If you're using Windows, there are Unix-like shells available for this platform too.

Of course, to follow along, you also need Vim or Neovim installed. Then, in your shell, you can simply run the command `vim` to open Vim, or `nvim` to open Neovim. You can also open a file if you give its filepath as argument; for example, `vim functions.lua`.

Finally, you'll need the book companion available on GitHub to follow along. You can clone it directly in your shell if you know how to do that, or you can download it directly from the GitHub repository (click on the `code` button, and then on `Download ZIP`).

That's it!

How To Get the Most Out of this Book

For this book not to fail your expectations, let me give you some recommendations here. Of course, you don't have to follow any of them: you can read this book from the first to the last page passively like a novel, read a page every three pages, or even toss it near your toilets to never open it again. It's up to you.

The most important advice: try to use the book companion in Vim, and experiment with all the commands and keystrokes we'll see in this book. This is called active learning: it's better than reading passively without trying anything in Vim. This is even better than doing the exercises.

You can also create your own cheatsheet, adding what you're learning as you go along. Again, it's different from getting any cheatsheet made by somebody else: it will be yours and, as a result, it will help you learn and consolidate your knowledge in your long term memory.

Writing has a powerful effect: it makes some external knowledge truly yours. I can't recommend it enough!

In that regard, writing not only a cheatsheet but a whole practice journal can be beneficial too. It's basically a journal helping you track your learning: what you're working on, or what you want to work on for example. You can write about the blockers you find in the way, the research you do on the side, and so on. You can also go back to your journal to refresh your memory.

You can even write your practice journal in Vim to create The Ultimate Feedback Loop of Learning©!

I'll never repeat it enough: try to have fun! Learning something shouldn't be a chore, and it's also true for Vim. You don't need to try to learn everything all at once. Only reading this book from 10 to 30 minutes a day can be enough; the important thing is to stay consistent. It's better to try to learn Vim for 20 minutes every two days, than 10 consecutive hours every two months.

Structure of the Book

The book is not divided in chapters, but in "ranks". It's because I wanted the book to have some sort of progression, from the basic concepts to more advanced ones. When you see that I speak about a "rank", think of it as a chapter.

Each rank build on the knowledge of the previous ones. That's why there will be more and more references to past ranks as you go throughout the book. I also refer to the future ranks quite often, for you to check them out at your leisure.

The first section of each rank will be about something big and important, and I'll often cover more than one mode there (we'll see Vim modes in rank I). The other sections will often be about one mode specifically, or about customizing Vim. This information is always included in the title of the section.

Notation Conventions

Vim has a powerful help system embedded in the editor; we'll come back to that in rank II. It uses specific notations for different ideas, and this book use most of them:

Notation	Description
`[]`	Placeholder for an optional part of a command.
`{}`	Placeholder for a required part of a command.
`<character>`	Key notation for a special character.
`CTRL-{character}`	Keystroke notation where you need to hold `CTRL` and hit `{character}`.
`CTRL-{character1} {character2}`	Keystroke notation where you need to hold `CTRL` and hit `{character1}`. Then, release `CTRL` and hit `{character2}`.
`{mode}_CTRL-{character}`	Keystroke notation for a specific `{mode}`.
`'number'`	Vim option

Let's look at some examples:

Example	Description
`:set {option}`	The argument `{option}` is mandatory.
`:[range]delete`	The prefix `[range]` is optional.
`<esc>`	The "escape" key.
`<enter>`	The "enter" key.
`CTRL-v`	You need to hit and hold the `CTRL` key first, and hit `v`.
`i_CTRL-v`	You need to hit and hold the `CTRL` key first, and hit `v` in INSERT mode.
`CTRL-w s`	You need to hit and hold the `CTRL` key first, and hit `w`. You can then release the `CTRL` key from your mighty pressure, and hit the `s` key.
`w`	You need to hit the `w` key.
`W`	You need to hit the `W` key (the uppercase of `w`).
`d$`	You need to hit the `d` key, and then the `$` key.
`:write<enter>`	You need to hit `:`, then the five letters `write` consecutively, and then the `ENTER` key.

In this book, the word "keystroke" means a set of key(s) which can be pressed at the same time, or consecutively.

We won't use often the keystroke notation including the mode in this book (for example `i_CTRL-v`). That being said, Vim help use the same notation, so it's useful to be aware of it if you want more information about a keystroke which is not in NORMAL mode.

Regarding the key notation for special characters, Vim help often mix uppercase and lowercase; for example, `<Esc>`, `<PageUp>`, or `<BS>`. It's annoying; that's why this book always uses lowercase for this notation. The case doesn't change anything anyway: `<ESC>`, `<Esc>` or `<esc>` are equivalent.

This book will also try to help you remember the different keystrokes you can use in Vim, by linking them with what they're doing. For example, to link "w" with "word", the "w" of "word" will be underlined as follows: "word". It's to remember that, when you hit "w" in NORMAL mode, you would move one "word" forward.

If you're new to Vim, you might have no idea what we're speaking about here. No worries: we'll look at all these concepts in more details down the line. Just remember that, if you have difficulties to understand the notations of this book, you can always come back here to lighten your burden.

Playing Vim: The Exercises

Here are some general rules regarding the many exercises scattered all over the book:

1. The exercises at the end of some sections are the easiest ones.
2. The exercises at the end of a chapter (rank) go a bit deeper in each topic.
3. The exercises in the section "beyond the rank" are harder. They often introduce complementary concepts from the ones seen in the current rank. You'll often need to look at Vim help to solve them.
4. The solutions come after the rank itself.
5. After finishing an exercise and before beginning another one, always undo your changes. If you have Git installed on your system, running the shell command `git checkout *` in the root directory of the book companion you've downloaded should be enough.

I tried to give the best solutions for each exercise, but it's likely there are other (and better) ones.

This book is full of information. I wouldn't advise you to try to remember everything, just what's useful for you. If you don't really know what's useful, the exercises at the end of each section can show you the most important points.

When it comes to notation for the exercises, in many of them you'll see a black square. It represents the cursor position. For example:

```
This is some text and the cursor is on the "t" of "text".
```

This square indicates where you should put your cursor before beginning the exercise. That said, your specific cursor might not be represented as a block ▉ in your terminal, but as a line `|`. In that case, simply put your cursor just before the letter highlighted. To come back to our example above, if your cursor looks like a line, you need to place it before the letter `t` of `text`:

```
This is some |text and the cursor is on the "t" of "text".
```

Enough rambling. Let's now begin our journey in Vim Land.

Becoming a Vim Player

Before diving into Vim itself, it's important to understand how Vim can help you in your writing or in your coding. It's also a good occasion for me to justify the weird title of this book.

Vim is an Instrument

It's how I see Vim: as an instrument. Like a musical instrument, it can be challenging to understand, but it won't get in the way to create what you want to create.

After all, a pianist doesn't actively think about all the notes available when playing. Instead, if she's sufficiently trained, her movements will be automatics, to some degree, allowing her to focus solely on the music. Vim also relies on your muscle memory for you to focus on the most important: you're writing.

The concept of a piano is simple to understand. If you hit a key, you'll hear a note. Yet, from these simple keys you can build chords, scales, and melodies. It's similar with Vim: it's quite easy to use, but hard to master. You'll be able to compose the basics command to create more complex and powerful ways to write and edit your texts.

Finally, like an instrument, Vim is fun to use. First, because you only need to use your keyboard to do so; no need of anything else, like the mouse. It allows you to focus on what you really want to do without your editor going in the way. It's rewarding and fulfilling.

The Power is in Your Fingers

As we just said, Vim allows you to write and edit your text only using your keyboard. No need to spend your time pointing and clicking with your mouse. Yet, it's normal to use a mouse to write some text for most (my younger self included). I'm now convinced (and I have the experience to back it up) that we don't need a mouse to write; only using the keyboard is easier. It's changing our habit which can be hard.

When you are in a state of flow, focused on your craft, you don't want to be interrupted every five minutes by moving your hand to the mouse, or by looking at your fingers. You shouldn't have to do any of that. You should only care about what you're creating.

Before being able to use a piano, you need to know how to place your hands properly on the keys to be as effective as possible. It's again similar with Vim: knowing how to type efficiently on a keyboard is a stepping stone to master a keyboard driven workflow.

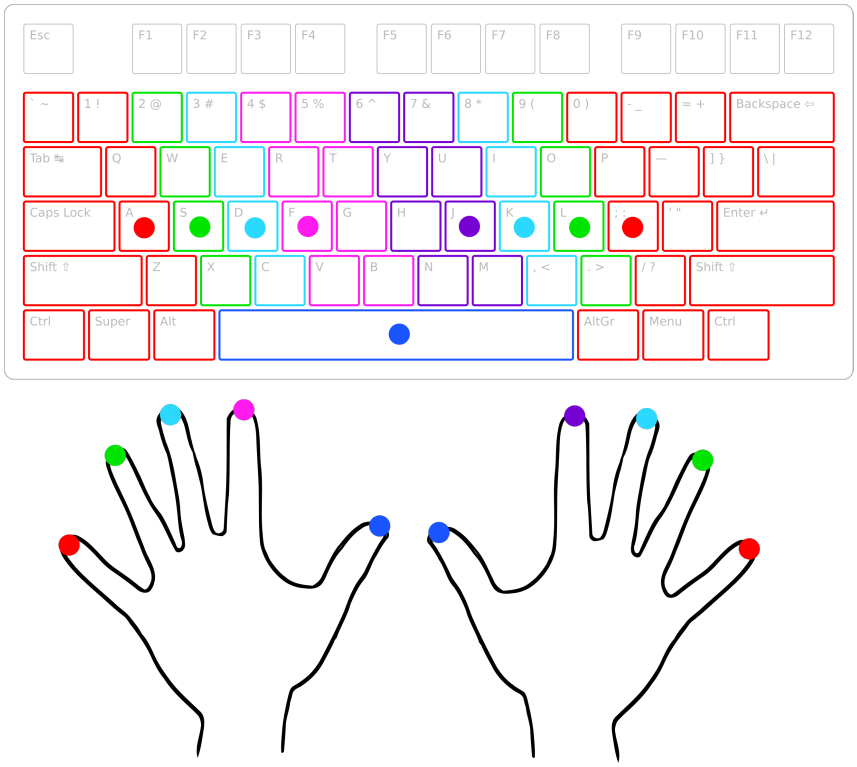
You can practice the techniques we'll see next while reading the rest of the book. It will be difficult at the beginning, but stick to it as much as you can. You won't regret it.

Efficient Typing: the Two Rules

It's satisfying to see your typing techniques improving days after days, months after months, and even years after years. Like Vim, these techniques are easy to learn but difficult to master.

The first rule for a good typing: placing your hands correctly. The keys `a`, `s`, `d`, `f` as well as the keys `j`, `k`, `l`, `;` are called the row keys. They are the starting points for your hands. From there, you'll be able to grab any other key as efficiently as possible.

You'll notice that there are little bumps on the keys `f` and `j` on your keyboard. They are indicators for you to know where you need to put your indexes. When they are at the good position, simply place the other fingers on the other row keys, as shown on the illustration below.



The second rule you need to train for: try not looking at your keyboard while you're typing. Of course, if you don't remember where a key is, look at it, but only after trying blindly where you think it is. We want to train your muscle memory here.

I was only typing with two fingers before trying to follow these two rules. It felt really weird at first; now, I wouldn't type differently. It's efficient, it's comfortable, it's great!

The First Week

When you decide to use the two rules we saw above, you need to try to follow them all the time. We need 100% commitment here. If you surprise yourself using your bad technique again, which will happen, don't worry: simply come back to the good one. This is part of the learning process, not a horrible failure cursing your whole family on five generations.

The first three days are the most difficult. You'll alternate between good and bad technique without even noticing it. You'll do mistakes. You'll be slower. That's great! It's how you'll learn.

Fortunately, at the end of the week, the amount of mistakes you'll make will decrease, and the need to watch the keyboard will slowly disappear.

The Second Week

You'll notice during the second week the amount of mistakes decreasing even more, and you won't dare look at your keyboard while typing anymore. At the end of the week, you'll see your typing speed improving already. A rewarding pleasure will begin to surge in your brain. That's what we all want.

Speed and Accuracy

During your two weeks of initial training, you shouldn't focus on speed or accuracy. Just type, as much as you can, and don't worry about anything else yet. Not even about the mistakes you're making.

Only then, when you feel comfortable enough, you can shift your focus on speed and accuracy: how fast you can type while making as few mistakes as possible.

Keyboard Layouts

You noticed that I'm only covering the US international keyboard layout here. If you use another one, the same principles apply. The row keys are at the same position too; only the keys themselves change.

A last tip: if you don't use your Caps Lock key often, try to remap it to something more useful. Vim users hit their Escape keys again and again, so it's a good candidate. Depending on the operating system you're using, you could also remap the Caps Lock key for acting as Ctrl when you hold the key, and Escape when you briefly hit it. It's how my Caps Lock key behave, and it's awesome. I can't recommend it enough.

Practice, Practice, Practice

As always, to learn as fast as possible, you need to practice. Again, this book ask you to use your keyboard extensively, so you'll have many occasions to practice.

You can also use some typing software to have concrete data about your speed and accuracy. Here are my favorites:

- Type Racer
- Online Typing Test WPM
- Speed Coder

Rank I – Rookie

This is where our adventure begins: on the shores of the most basic functionalities Vim has to offer. If you try to actively understand the concepts explained in the first two ranks, you'll be able to use Vim for your writing. It won't replace your favorite text editor or your IDE right away; but it will be enough to edit simple text files (like configuration files, for example).

We'll see, in this chapter:

- The basic Vim modes. This is primarily what makes Vim so different from any other text editor.
- How to move your cursor around, only using your keyboard.
- How to use the "language of Vim". Said differently, what are operators, motions, and text-objects.
- How to undo and redo your changes.
- How to configure Vim.

I recommend you to open the file "functions.lua" in Vim from the book companion and try to experiment with the different examples given here. To do so, move to the root directory of the book companion, and run `vim functions.lua` in your shell (or `nvim functions.lua` if you use Neovim).

I see that you're bursting with anticipation: let's not wait any longer.

Vim: a Modal Editor

Vim is a modal editor: depending on the mode you're in, the different keys you'll hit on your keyboard will have different effects. This is the first big difference with more mainstream text editors, and arguably one of the biggest reason why Vim is so beloved and so powerful.

The two modes you'll use the most are:

1. The NORMAL mode.
2. The INSERT mode.
3. The COMMAND-LINE mode.

It's important to understand their purposes to use Vim efficiently.

The NORMAL Mode

Normally, after opening a text editor, you can directly type anything you want to see your writing appearing on the screen. Nothing more basic; yet, it doesn't work like that in Vim. It's why many beginners are confused when they try to use Vim for the first time.

To understand what I mean, try to open Vim. You'll see a welcome message. Now, if you try to hit the letter "x" on your keyboard, no "x" will appear on the damn screen.

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

To really understand what I mean, it's best to begin by what you already know. Let's say that you want to replace a portion of some existing text using a more mainstream text editor. To do so, you would often need to:

1. Use the mouse and move your pointer to the portion of text you want to edit.
2. Click on your mouse to move your cursor (or to select some text).
3. Write whatever you want with your keyboard.

This way of working is quite intuitive, but not very efficient. If you know your editor well, you might also know some keyboard shortcuts to increase your efficiency. For example, in many text editors out here, instead of using your mouse to move your cursor, you could:

1. Use a keyboard shortcut (like `CTRL-f` for example) to search the word you want to edit.
2. Delete the word with the `DELETE` (`<del>`) key.
3. Write whatever you want with your keyboard.

Why using keyboard shortcuts is more efficient? Even if we still need three steps to edit our text, we don't need to use the mouse in the second example. It's better for your hands (nobody wants carpal tunnel syndrome), it's more comfy, and it's also more efficient.

This is basically the purpose of Vim's NORMAL mode: because it's not used to write any text, every single key on your keyboard is a keyboard shortcut, offering you many different commands to move your cursor and target the exact piece of text you want to edit.

Said differently, Vim's NORMAL mode is a keyboard-centered way to control your editor, *telling* Vim what you want to do, and it will obey your mighty will. You'll soon become a God (or Goddess), and Vim will be your slave.

But if there are even more NORMAL mode commands than keys on a keyboard, how is it possible to remember them all? Believe it or not, 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 build some muscle memory. Even better: they are composable; you can combine some of them in a logical, easy-to-remember way. A bit like you would use different notes on a piano to create a melody.

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

Enough theory and metaphors: it's time to practice. We first need some text to edit, so let's insert some text first. To do so, let's switch to the second most important mode, the INSERT mode.

The INSERT Mode

Let's now hit our first NORMAL mode command, which will switch Vim from NORMAL mode 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 `-- INSERT --` in the bottom left corner of Vim.

Welcome to INSERT mode!

You're now able to type the text you always wanted to bring on your screen. Go ahead, don't be afraid: type anything you want, like you would do in any other editor. At the end, Vim is not *that* different.

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

Welcome back to NORMAL mode!

That's exactly what do a Vim user many times during a writing session: switching between NORMAL mode to edit existing text, and INSERT mode to insert new text.

The NORMAL mode command `a` can also let you switch to INSERT mode, but it will do it after the character you're on. Try it to see the difference! Remember: to switch back to NORMAL mode, simply hit `ESCAPE`.

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

To summarize what we've just seen:

Keystroke	Description
`i`	Switch from NORMAL mode to INSERT mode.
`a`	Switch from NORMAL mode to INSERT mode after the character under the cursor.
`<esc>`	Switch back from another mode to NORMAL mode.

You can see here that I use the notation `<esc>` representing the `ESCAPE` key. As we saw in the preface, keystrokes with special characters in this book are surrounded with `<>`.

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

The COMMAND-LINE Mode

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

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

In NORMAL mode, you can hit NORMAL mode commands; in COMMAND-LINE mode, you can execute *Ex commands*. The word "command" is used differently here depending on the mode, and, as a result, this word becomes quite confusing. That's why I'll speak about NORMAL mode keystrokes, instead of NORMAL mode commands in this book. That said, be aware that many other resources about Vim (including Vim help) often speak about NORMAL mode commands.

Let's go back to our new mode, the COMMAND-LINE mode. First, to switch to COMMAND-LINE mode, you need to use the NORMAL mode keystrokes `:`. To come back to normal mode, you need again to hit the `<esc>` key.

When you switch to COMMAND-LINE mode, your cursor moves automatically at the very bottom of Vim, just after a colon `:`, indicating that you can write and run an Ex command.

You can think of Ex commands as the menu you would normally go into in a more mainstream text editor, often using your mouse. For example, you can use the COMMAND-LINE mode to save your file, or to search and replace some text.

Here are some basic ones:

Ex command	Short name	Description
`:write`	`:w`	To write (save) the current open file.
`:edit {filepath}`	`:e {filepath}`	To edit the file located at `{filepath}`.
`:quit`	`:q`	To quit the current window.
`:quit!`	`:q!`	To quit the current window without saving!.

To quit Vim, you need to quit all windows. We'll see more about windows in rank III.

You can see that most Ex commands have both long names (like `:write`) and short names (like `:w`). They do the same things; the long version is easier to understand, but the short version is faster to type.

A last important Ex command, maybe the most important of all:

Ex command	Short name	Description
`:help {subject}`	`:h {subject}`	Open Vim's help about `{subject}`.

For example, if you want to know more about the NORMAL mode command `i`, you can run the Ex command `:help i`. It will open a new window with the information you seek.

Do you already have a best friend? Ditch her. Vim help is your new best friend from now on. It's you're go to if you need any kind of information about anything Vim, really. If you don't remember how to quit Vim for example, you can run the Ex command `:help quit`.

I'll often reference Vim help in this book, at the end of most sections. It will allow you to dig deeper into the functionalities we'll cover.

For example:

Help Yourself

```
:help vim-modes
:help write-quit
:help cmdline-completion
```

Don't worry if you don't really understand Vim help at first, or if there is too much information. The more you'll get comfortable with Vim, the more it will make sense.

Here's an important tip when using the COMMAND-LINE mode: you can use the `<tab>` key to complete Ex commands. For example, if you type `:wr` in COMMAND-LINE mode followed by

`<tab>`, Vim will complete the Ex command `:write` for you. It's useful when you don't remember the exact Ex command, or to discover new ones.

Also, you can use the keystroke `CTRL-d` to display the possible completions.

Similarly to a shell (like Bash or Zsh), you can also use the arrow keys `<up>` and `<down>` to go through your Ex command history.

It's Playtime!

Exercise A – Vim, a Modal Editor

Open Vim in your terminal. You should go through the steps below one after the other, keeping the changes you've made at each step.

1. Insert the text `Hello Vim Land`, and come back to NORMAL mode.
2. Insert a bang `!` after the word `Land`, and come back to NORMAL mode.
3. Search in Vim help for "vim-modes".
4. Quit Vim without saving.

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 you'll see in this first rank. You can always come back to it and experiment with the ones you forgot.

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 around.

As we saw already, our fingers should stay on the row keys of the keyboard while using Vim, for two reasons:

1. For our typing speed and accuracy to improve.
2. Because the Vim's keystrokes you can use in NORMAL mode are more easily accessible if your hands stay on the home row.

Your hands shouldn't move too much; only your fingers should. If you look at your arrow keys, you'll see that they're far away from the row keys, forcing you to move your right hand 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 strongly 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 historical reasons. Vim is the successor of the text editor Vi, which was used on physical terminals. When you look at the

keyboard of some of them (like the Lear Siegler ADM 3A terminal for example), you'll see that the arrow keys are 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, let's try to answer an important question: how to remember what `h`, `j`, `k` and `l` do in NORMAL mode? Here are some useful mnemonics:

1. 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.
2. The `j` key moves your cursor down. Here are 3 mnemonics you can try to remember:
   - On your keyboard, the `j` key has a little bump *at the bottom*; so `j` moves the cursor down.
   - With some imagination, the letter "j" looks a bit like "↓".
   - Let's speak typography: the letter "j" has a descender, meaning that part of the letter descends from its baseline. As a result, `j` "descends" your cursor.
3. The `k` key is the only one left, so it has to go up. To come back to the secret art of typography, the letter "k" has an ascender, meaning that part of the letter ascend from its baseline. So `k` "ascends" our cursor.

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 have to use `hjkl`. If you prefer puzzle games, try this wonderful sokoban.

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.

Here are the most useful of these motions:

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.
`ge`	Move backward to the end of the previous 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, a "word" doesn't include some special characters.

For example, you can open the file "functions.lua" from the book companion, and place your cursor at the beginning of the following line:

```
local function restorePosition() {
```

Now, try to use the motions `w` and `W` to see the difference. The "WORD" motion will skip the parenthesis, but the "word" motion won't.

There are even 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 the 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, Middle, and End of Line

If you want to go to the beginning or the end of the current line, moving word by word can get boring quickly. Here are some more NORMAL mode keystrokes which will help you:

Motion	Description
`0`	To move to the first character of the current line.
`$`	To move to the last character of the current line.
`^`	To move to the first non-whitespace character of the current line.
`gM`	To go to the middle of the current line.

In Vim, a whitespace can be a `<space>` or a `<tab>`.

It's Playtime!

Exercise B – Horizontal Motions

Open the file "functions.lua" from the book companion. Using the `hjkl` keys in NORMAL mode, move your cursor to the following position:

```
#/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
```

You should go through the steps below one after the other, keeping the changes you've made at each step.

1. Move to the first `f` of `function` using one key.
2. Move back to the first `r` of `restorePosition()` using one key.
3. Move to the end of the line using one key.
4. Move to the beginning of the line using one key.
5. Move to the first `s` of `restorePosition` using two keys.

Vertical Motions

Moving our cursor on the same line is great. But, at one point, we'll have to go up and down too. Here are some more NORMAL mode keystrokes to move our cursor vertically:

Motion	Description
`{line_number}G`	Move at the beginning of the `{line_number}`.
`1G` or `GG`	Move to the first line.
`G`	Move to the last line.
`CTRL-u`	Move upward half a screen.
`CTRL-d`	Move downward half a screen.
`CTRL-b`	Move backward (upward) an entire screen.
`CTRL-f`	Move forward (downward) an entire screen.

For example, the keystroke `10G` will move your cursor on line 10.

You can also use the 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 more keystrokes allowing you to move to the top, middle, or to the bottom of the current window:

Motion	Description
`H`	Move to the first line (the highest line) of the screen.
`M`	Move to the line at the middle of the screen.
`L`	Move to the last line of the screen.

Help Yourself

```
:help up-down-motions
```

It's Playtime!

Exercise C – Vertical Motions

Open the file "functions.lua" from the book companion. Using the `hjkl` keys in NORMAL mode, move your cursor to the following position:

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

You should go through the steps below one after the other, keeping the changes you've made at each step.

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 to the middle of the screen.

Undo and Redo (NORMAL mode)

Personally, I would be in great difficulty if I didn't have any way to undo or redo my editing. Fortunately, there are some NORMAL mode keystrokes we can use to come back in time:

Keystroke	Description
`u`	To undo the last edit.
`CTRL-r`	To redo the last undo.

You can think of `CTRL-r` as you being in control (`CTRL`) of your text. It's not the perfect mnemonic, but it was good enough for me.

You'll notice that switching to INSERT mode, writing some text, and coming back to NORMAL mode is only one undo node. It means that everything you've inserted can be erased with one

hit on `u` after coming back to NORMAL mode.

In Vim, the undo mechanism is way more powerful than in most other text editors. We'll learn more about it in rank XI.

Help Yourself

```
:help undo-redo
```

Operators, Motions, and Text-Objects (NORMAL Mode)

Some NORMAL mode keystrokes can be seen as notes. We can compose a melody from these keystrokes, mainly to edit our text. We can also see these keystrokes as verbs or nouns, creating a tiny Vim language.

My unwanted opinion: it's nothing less than brilliant.

The Operators

We've seen how to move our cursor in Vim with motions. But learning to walk is only the beginning; now, it's time to *do* something in NORMAL mode. To operate on our text.

The operators are the verbs of our Vim language. Here are the most useful ones:

Operator	Description
`d`	To delete some text.
`c`	To change some text.
`y`	To yank (copy) some text.

To be even more specific, when you ask Vim to "change" some text, it simply deletes it and then switch automatically to INSERT mode. You can then type your changes.

If you try to hit `d`, `c`, or `y` in NORMAL mode, nothing will happen. You always need to combine operators with something else; with themselves, for example:

Keystroke	Description
`dd`	To delete the current line.
`cc`	To change the current line.
`yy`	To yank the current line.

That's not all: you can also combine operators with motions.

Operators and Motions

Here are examples of operators combined with some motions we've seen above. You can hit these keystrokes in NORMAL mode:

Example	Description
`d$`	To delete the text from your cursor to the end of line. Equivalent to the alias `D`.
`c$`	To change the text from your cursor to the end of line. Equivalent to the alias `C`.
`y$`	To yank the text from your cursor to the end of line. Equivalent to the alias `Y` (only in Neovim by default).
`cw`	To change from your cursor to the end of the word.
`yG`	To yank (copy) from your cursor to the end of the file.

Remember that "yank" is a synonym of "copy" in Vim. We'll "yank" a lot in this book, and Vim help also uses this word a lot. Soon, you'll also use it with your family and your friends, potentially losing them in the process.

These operators (especially the yank operator) can be even more useful when you know the following NORMAL mode keystrokes:

Keystroke	Description
`p`	To put (paste) the last yanked or deleted text *after* the character under your cursor. If you yank or delete an entire line, `p` will put it after the current line.
`P`	To put (paste) the last yanked or deleted text *before* the character under your cursor. If you yank or delete an entire line, `P` will put it before the current line.

I encourage you, in Vim, to combine operators, motions, and text-objects, as well as trying to understand how the put keystroke behave. Again, the more practice you'll have, the more proficient you'll get!

Help Yourself

```
:help operator
:help objet-motions
```

Operators and 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 can be seen as nouns.

When you use a motion with an operator, you'll operate on the text from your cursor position until the end of the motion. A text-object is a set of characters with a specific, determined start and end, and the start is not necessarily your cursor position.

In Vim, "a word" is a text-object, as well as "a sentence", or "a paragraph". Let's look at some examples of operators combined with text-objects:

Example	Description
`diw`	To delete inside the word. It deletes the current word under the cursor.
`daw`	To delete around the word. It deletes the current word under the cursor, as well as the whitespace following it.
`ciw`	To change inside the word. It deletes the current word under the cursor and switch to INSERT mode.
`dip`	To delete inside the paragraph.

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 help, `daw` is described as delete a word. I find this "translation" quite confusing, because it doesn't only delete the word, but also the following space; that's why I use around instead of a.

In general, text-objects beginning with a "a" (like `aw`) often delete something more than the "object" itself.

There are even more text-objects available in Vim for your editing needs.

Help Yourself

```
:help text-objects
```

It's Playtime!

Exercise D – Operators and Text-Objects

Open the file "functions.lua" from the book companion. 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
```

You should go through the steps below one after the other, keeping the changes you've made at each step.

1. Delete the word `local` in one operation, keeping the space after it.
2. Undo what you just did.
3. Delete the word `local` in one operation, including the space following it.
4. Delete the word `function`, and directly switch to INSERT mode, in one operation. Then, come back to NORMAL mode.
5. Undo all your changes.
6. Move to the `a` of the word `local` using two keys, and hit `dw`. What's the difference between `dw` and `diw`?

Bending Vim to Your Will (Customization)

In Vim, many functionalities are configurable; you can shape your editor according to your megalomaniac desires. Let's begin here with the basics.

You can find the final configuration we'll write in this section in the book companion. You can use both Vimscript and Lua implementation as references.

The Main Configuration File: the vimrc

Your main configuration file can be in the following path, depending on what editor you use:

Editor	File	Config language
Vim	`~/.vim/vimrc`	Vimscript
Neovim	`$XDG_CONFIG_HOME/nvim/init.vim`	Vimscript
Neovim	`$XDG_CONFIG_HOME/nvim/init.lua`	Lua

When Vim starts, it reads the configuration file `vimrc` (or `init.vim`, or `init.lua`), and each line is executed. 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 `/home/user/.config`, depending on your operating system. If you don't know what are the XDG user directories, here's a good resource to learn more about them.

If you use Neovim, you can also write your configuration in Lua (instead of Vimscript) using the file `init.lua`. You can't use both `init.vim` and `init.lua`; you need to make a choice.

Personally, I use Neovim, and I also like to use Vimscript as much as possible (it's less verbose than Lua in many cases), only using Lua for more complex pieces of configuration. As a result, I write my configuration in Vimscript using `init.vim`, and I load directly from there some Lua files when I feel that using Lua is necessary.

We'll dive more into these ideas in rank II. Whatever the vimrc you decide to use in Neovim (`init.vim` or `init.lua`), just remember that you don't have to configure everything in one language; you can use both Vimscript and Lua.

Help Yourself

`:help vimrc`

First Configuration

Let's now write our first lines of configuration. I would encourage you to write them using Vim. Even if you're a total beginner, what we've seen 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 ~/.vim/vimrc` (or `nvim $XDG_CONFIG_HOME/nvim/init.vim` if you use Neovim).

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

```
" Disable the arrow keys (use hjkl instead, respectively)
nmap <left> <nop>
nmap <down> <nop>
nmap <up> <nop>
nmap <right> <nop>
```

Don't forget to run the Ex command `:w` in COMMAND-LINE mode to save your changes.

In Vimscript, any line following a double quote `"` is a comment. It means that Vim will never try to execute the line. We can use comments to add some explanations to our configuration.

The Ex command `nmap` allows us to create a mapping. We'll look at it more in details in rank III. Here, we map the arrow keys to... nothing, to use the row keys `hjkl` instead.

If you want to write your config in the vimrc `init.lua` for Neovim, here's the equivalent in Lua:

```lua
-- Disable the arrow keys (use hjkl instead, respectively)
vim.keymap.set('n', '<left>', '<nop>')
vim.keymap.set('n', '<down>', '<nop>')
vim.keymap.set('n', '<up>', '<nop>')
vim.keymap.set('n', '<right>', '<nop>')
```

To experience your new configuration, you can relaunch Vim and try to use your arrow keys; they shouldn't work anymore. Great! Out of some constraints can come great creativity.

Clipboard Management

Copy-pasting text from another application to Vim and vice-versa is a simple operation beginners often take for granted. But it comes with surprising difficulties, especially if you use Vim (instead of Neovim).

The Clipboard Functionality

If you're using Neovim, you can skip this subsection.

First, let's try to run the following in the terminal:

```
vim --version
```

You'll see something like that (among other information):

```
+channel          +ipv6             +persistent_undo
+cindent          +job              +popupwin
-clientserver     +jumplist         +postscript
-clipboard        +keymap           +printer
```

Any feature prefixed with a plus `+` is compiled with your version of Vim, and anything with a minus `-` is not. As you can see in the example above, my Vim has not the `clipboard` feature, meaning that Vim won't be able to store information in my operating system's clipboard, or retrieve some information from it. That's a bummer.

If you're using Vim, make sure that you have the `clipboard` feature. You can look at your package manager to see how you can install Vim with it. For example, in Arch Linux, you'll have to install the package `gvim`, which also install Vim for the shell, compiled with the clipboard feature.

Using your Operating System's Clipboard

Let's add another line to our vimrc:

```vim
" Can copy-paste more easily from and to Vim
set clipboard+=unnamedplus
```

In Lua:

```lua
-- Can copy-paste more easily from and to Vim
vim.opt.clipboard:append({'unnamedplus'})
```

It will make the copy-paste mechanism less confusing. We'll look at this more in details when we'll look at registers in rank IV.

Help Yourself

`:help clipboard`

Improving Vim's Defaults

We now have Ex commands in our vimrc. That's right: you could also run each line we've written in COMMAND-LINE mode. For example:

```
:nmap <up> <nop>
```

If you use Neovim and you want to run some Lua directly from COMMAND-LINE mode, you can use the Ex command `:lua` before your Lua snippet as follows:

```
:lua vim.keymap.set('n', '<up>', '<nop>')
```

That said, if you only use the above Ex commands directly in Vim without writing them in your vimrc, these new mappings would disappear when you close Vim. That's why we write them in a vimrc; because we want these Ex commands to be executed each time we open Vim.

If you use Vim instead of Neovim, let's add these lines to your vimrc too:

```
" No compatibility with Vi
set nocompatible

" Enhanced completion in command-line mode
set wildmenu

" Syntax highlighting
syntax on

" Enable filetype, indentation, plugin
filetype plugin indent on

" Always display the status bar
set laststatus=2

" Allow hidden buffers
set hidden
```

We basically set options to make Vim a bit more user friendly. We'll look more closely at Vim's

options in rank II. We're also looking at the option `'hidden'` more thoroughly in rank IV.

By default in Neovim, all of these options are already set as above.

Also, if you want to see the line numbers (in Vim or Neovim), you can add the following to your vimrc:

```
" Display line numbers
set number
```

In Lua:

```
-- Display line numbers
vim.opt.number = true
```

The Configuration Addiction

At that point in our adventure, 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 many hours trying to come up with the best configuration in the known universe.

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 worse, your IDE. You'll get eventually there when we'll speak about external plugins a bit later in this book but, before that, you should consider trying to understand and use the functionalities directly available in vanilla Vim.

There is a massive time sink black hole in Vim Land called The Pit of Endless Configuration™. It goes as follows:

1. You discover how configurable Vim is.
2. You spend a crazy amount of time configuring Vim and adding more and more plugins (instead of learning its fundamentals).
3. You don't understand what's happening in your growing vimrc anymore.
4. Vim begins to behave weirdly and, since you don't understand your own vimrc, you don't really know why.
5. You're burned out. Vim is hell, and you're back on Notepad.

Now, we all descend in the Pit of Endless Configuration at some point. There are so many articles out there telling us what configuration to write (without necessary explaining what it does), and there is this weird appeal of trying to make Vim perfect.

Vim is a tool. What you produce with it is the most important; not its configuration. Granted, it can be fun to configure Vim, like you would tune your instrument before playing it. But it can get also quite overwhelming.

Debugging Your Configuration

As we just saw, the more we add to Vim, the more we increase the chances to get some nasty bugs; especially if we don't really understand what we add.

If Vim becomes unstable and buggy, the first step is to try to disable what you've added, to understand where the problem is coming from.

These options can help you in that regard. They can be given to both Vim and Neovim:

Shell command	Description
`vim -u NONE`	Launch Vim without your vimrc.
`vim --noplugin`	Launch Vim without your plugins.

If you find out that your misery comes from your vimrc, try to comment out the last additions to pinpoint the problem. You can also try to disable the different plugins you've installed recently if the problem comes from there.

We'll look more thoroughly into Vim plugins in rank TODO.

Exercises

To solve these exercises, open the file "functions.lua" from the book companion in Vim.

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

Exercise 1 - Horizontal Motions

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
```

You should go through the steps one after the other, keeping the changes you've made at each step. Use the most efficient keystrokes you know in NORMAL mode; the less keys you hit, the better:

1. Move your cursor to: `if vim.fn.line("'\"") > 1`
2. Move your cursor to: `if vim.fn.line("'\"") > 1`
3. Move your cursor to: `if vim.fn.line("'\"") > 1`
4. Move your cursor to: `if vim.fn.line("'\"") > 1`

Exercise 2 - Operators, Motions, 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
```

You should go through the steps one after the other, keeping the changes you've made at each step.

1. Move to the first opening parenthesis of the same line, using only two keys, as follows:

```
if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
```

2. Delete these parentheses and everything inside as follows:

```
if vim.fn.line > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
```

3. Undo your previous deletion.
4. Move your cursor to the `i` of the first word `line`, using only two keys, as follows:

```
if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
```

5. Move to the `i` of the first `vim`, using only a one key, as follows:

```
if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
```

Exercise 3 - Yank and Put

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
```

You should go through the steps one after the other, keeping the changes you've made at each step.

1. Hit a single key in NORMAL mode to move to the end of the line and switch to INSERT mode. Come back to NORMAL mode afterward.
2. Yank the whole line.
3. Put the line you've yanked above the current one, using a NORMAL mode keystroke.

Beyond the Rank

These exercises are more difficult. The solutions will often involve some complementary concepts not seen in this rank.

Exercise 4 - Yank, Delete, and Put

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
```

You should go through the steps one after the other, keeping the changes you've made at each step.

1. Yank the whole line.
2. Move to the first line of the file using two keys.
3. Delete the whole line.
4. Put the line you've deleted above the current one.
5. Put the line you've yanked below the current one.
6. Put the line you've deleted above the current one, using an Ex command instead of a NORMAL mode keystroke.

Exercise 5 - 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
```

For each step below, undo all your changes and come back to this initial cursor position.

1. Delete the whole function `restorePosition` (spreading on 5 lines), as well as all the empty lines after it, only using three keys.
2. Delete the next block of parenthesis, only using three letter keys.
3. Delete a sentence.
4. To find the start and end of the text-object representing a 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 6 - More Operations

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

For each step below, undo all your changes and come back to this initial cursor position.

1. Delete the letter `r`, only using two keys, as follows:

```
local function estorePosition()
```

2. Delete the space before your cursor, only using two keys, as follows:

```
local function restorePosition()
```

3. What are the keystrokes of the two previous questions: operators combined with motions, or operators combined with text-objects?
4. We used two keys for each solution of question 1 and 2. Find an equivalent keystroke for each, only using one key this time.
5. What are the differences between the text-objects “ap” and “ip”?

Exercise 7 - 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
```

You should go through the steps one after the other, keeping the changes you’ve made at each step.

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.

Exercises - Solutions

Other solutions than the one presented here are possible.

Exercise A - Vim, a Modal Editor

Question	Keystroke	Result
start		H
1	`iHello Vim Land<esc>`	Hello Vim Land
2	`a!<esc>`	Hello Vim Land!

3. You need to run the Ex command `:help vim-modes`.
   First, hit `:` in NORMAL mode to switch to the COMMAND-LINE mode.
   Then, write `help vim-modes` and hit `<enter>`.
4. You need first to run the Ex command `:quit` (or `:q`) to quit the window opened by Vim help, and then you need to run `:quit!` (or `:q!`), to quit the last window without saving your text. Closing all windows will also quit Vim.

Exercise B - Horizontal Motions

Question	Keystroke	Result
start		`local function restorePosition()`
1	`b`	`local function restorePosition()`
2	`w`	`local function restorePosition()`
3	`$`	`local function restorePosition()`
4	`0`	`local function restorePosition()`
5	`fs`	`local function restorePosition()`

Exercise C - Vertical Motions

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

You can also run `:5` in COMMAND-LINE mode, to move to the fifth line of the file.

Exercise D - 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<esc>`	`restorePosition()`
5	`uu`	`llocal function restorePosition()`
6	`fadw`	`locfunction restorePosition()`

6. The difference between `dw` and `diw`:

- `w` is a motion; the operation `dw` begins at the cursor position.
- `iw` is a text-objects; the operation `diw` begins at the start of the text-object.

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

Also, a text-object always need to be prefixed by an operator. A motion can be used on its own to move your cursor.

Exercise 1 - Horizontal Motions

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

Exercise 2 - Operators, Motions, and Text-Objects

Question	Keystroke	Result
start		`if vim.fn.line("'\"") > 1`
1	`f(` or `t"`	`if vim.fn.line("'\"") > 1`
2	`da(`	`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 3 - Yank and Put

Question	Keystroke	Result
start		`local function restorePosition()`
1	`A<esc>`	`local function restorePosition()`
2	`yy`	`local function restorePosition()`
3	`P`	`local function restorePosition()`

Exercise 4 - Yank, Delete, and Put

Question	Keystroke	Result
start		`local function restorePosition()`
1	`yy`	`local function restorePosition()`
2	`gg` or `1G`	`#!/usr/bin/env lua`
3	`dd`	
4	`P`	`#!/usr/bin/env lua`
5	`"0p`	`local function restorePosition()`
6	`:put!`	`#!/usr/bin/env lua`

Exercise 5 - More Text-Objects

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

Note that `dab` here is for delete around a block. A block includes the pair of parentheses and what's inside.

There's also `aB` and `iB`, both text-objects representing a block delimited with curly brackets `{}`.

4. You need to run the Ex-command `:help sentence` (or `:help text-object`) in COMMAND-LINE mode.
5. The text-objects `at` and `it` stand for "around a HTML tag" and "inside a HTML tag", respectively. Add an operator as a prefix to delete, change, or yank your HTML more easily.

Exercise 6 - More Operations

Keystroke	Keystroke	Result
start		`local function restorePosition()`
1	`dl`	`local function estorePosition()`
2	`dh`	`local functionrestorePosition()`

3. We're using motions here: `l` is the motion to move your cursor one character to the right, `h` to move your cursor one character to the left. Also, the motion `yl` is occasionally useful if you want to yank an exotic unicode character to put (paste) it somewhere else.
4. The NORMAL mode keystroke `x` is the equivalent of `dl`, and `X` is the equivalent of `dh`.
5. The text-object `ap` includes the blank line following the paragraph, `ip` doesn't. It's similar to the difference between `aw` and `iw`: the first includes the following space, but the second doesn't.

Exercise 7 - Up and Down Following Indentations

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

Rank II – Novice

Welcome back, Vim explorer. We just leveled up to a new rank! How do you feel? Good? Tired? Happy? Flabbergasted?

It's time to add more Vim modes to our tool belt, as well as diving deeper into the concepts we saw in the first rank. More specifically, we'll see:

- What are the VISUAL and REPEAT modes, and how to use them.
- More useful keystrokes to switch from NORMAL mode to INSERT mode.
- More keystrokes to delete some text, and the possible consequences.
- How to use efficiently your new life savior, Vim help.
- What to use for configuring Neovim: Vimscript or Lua.

Take your bag full of modes, motions, operators, text-objects, and let's continue our adventure in the Holy Land of Vim.

Even More Vim Modes

Vim's modes are like mountains: they're quite easy to understand (it's a big pile of rocks), but you can always dig deeper. To continue in my weird analogy, the NORMAL mode is the Everest of all modes: it's the biggest of all. We'll dig it a bit more in this section, and, in general, in the whole book.

But first, let's try to ascend two new modes: the VISUAL mode, and the REPLACE MODE

The VISUAL Mode

There is another important and useful mode in Vim: the VISUAL mode. Its goal? Selecting some text.

Here are the keystrokes you can use in NORMAL mode to switch to VISUAL mode:

Keystroke	Description
`v`	Switch to VISUAL mode "charwise".
`V`	Switch to VISUAL mode "linewise".

When you switch to VISUAL mode (similarly to INSERT mode), you'll see the indicator `--VISUAL--` appearing at the bottom left corner of Vim. As always, you can use the `<esc>` key to come back to NORMAL mode.

When hitting `v` in NORMAL mode, you enter the VISUAL mode per character ("charwise"). The selection will start at the cursor position; you can then hit some motions (or text-object) to select one or more characters.

When you have some text selected, you can then hit an operator to operate on your selection. Here are some examples:

Example	Description
`vaw`	Select visually around a word.
`vawd`	Select visually around a word, delete it, and come back to NORMAL mode. Equivalent to the NORMAL mode keystroke `daw`.
`vw`	Select visually from the cursor position to the end of the current word.
`v$`	Select visually from the cursor position to the end of line.
`vf,y`	Select visually from the cursor position to the first comma, and then yank the selection.
`vawyp`	Select visually around a word, yank it, and put (paste) it after the current character.

If you want to select entire lines at once, you can switch to VISUAL mode per line ("linewise"). To do so, you need to hit `V` in NORMAL mode, and then hit some vertical motions to add lines to your selection.

Here are more examples:

Example	Description
`Vy`	Select the current line and yank it. Similar to the NORMAL mode keystroke `yy`.
`Vj`	Select the current line and the line below.
`Vc`	Select the current line and change it. Equivalent to the NORMAL mode keystroke `cc`.
`VGd`	Select every line from the current one until the last, and delete the text selected.
`Vyp`	Select the current line, yank it, and put (paste) it below.

There is also the VISUAL mode per block ("blockwise"), but we'll see that later in the book, in rank VIII.

The different types of VISUAL modes are convenient because they allow us to operate upon an arbitrary portion of text we can visually select. Also, its visual nature make it easy to know what our operators will operate on.

That said, it's often quicker to only use operators with motions and text-objects in NORMAL mode (as we saw in rank I). For example, the operation `daw` is quicker to type than the equivalent `vawd`, because it involves less keys.

Editing your text in Vim while using the less keystrokes possible is a challenge Vim users like to tackle. There's even an excellent game based on this concept: VimGolf. Editing while typing less is often easier and faster.

This idea is powerful, but here's what you should really keep in mind: it's better to follow a workflow you like, you can remember easily, and which answers your specific needs, instead of obsessing on using the less keystrokes possible. If you can't remember (or if you don't like) the optimal way, go for the non-optimal route. That's fine too.

All in all, VISUAL mode is great to understand the range of motions and text-objects, even if it's not always the most efficient option to edit your text.

Help Yourself

`:help visual-mode`

The REPLACE Mode

The last mode I'd like to highlight here is the REPLACE mode. As you might have guessed, it's a mode where you can replace some text. Here are the keystrokes you can hit in NORMAL mode to use it:

Keystroke	Description
`r`	Replace the character under the cursor.
`R`	Switch to REPLACE mode.

Granted, the keystroke `r` doesn't switch to REPLACE mode, but it's thematically close.

As usual, after switching to REPLACE mode, you can come back to NORMAL mode by hitting `<esc>`.

In REPLACE mode, instead of inserting new text (what you can do in INSERT mode), you can replace existing text. It's especially useful when you don't want to mess up with some formatting.

Consider the following:

```
Stuff         Description
------------  --------------------------------
`r`           Replace the character under the cursor.
`R`           Switch to replace mode.
------------  --------------------------------
```

This is a table in markdown (you can find it in the book companion). If we want to change the word `Stuff` under the cursor by `Keystroke`, we could hit `caw` and then type `Keystroke` in INSERT mode. But the word `Description` would be misaligned afterward:

```
Keystroke         Description
------------  --------------------------------
```

Instead, if we hit `R` and type `Keystroke`, the word `Description` wouldn't move, because we would replace `Stuff` *as well as some following spaces* with our new characters:

```
Keystroke      Description
-------------  --------------------------------
```

Help Yourself

`:help replace-mode`

More Keystrokes to Switch to INSERT Mode

As we saw in rank I, using Vim requires you to switch often between NORMAL and INSERT mode. That's why there are many NORMAL mode keystrokes allowing us to switch to INSERT mode in slightly different ways.

Here are the most interesting ones:

Keystroke	Description
`i`	To insert before the current character.
`a`	To insert after the current character.
`A`	To insert after the end of the current line.
`o`	To open a new line below the current one and switch to INSERT mode.
`O`	To open a new line above the current one and switch to INSERT mode.
`<esc>` or `CTRL-c` or `CTRL-[`	Switch back to NORMAL mode.

Help Yourself

`:help insert-mode`

It's Playtime!

Exercise A – Even More Vim Modes

Open the file "functions.lua" from the book companion. 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
```

For each step below, undo all your changes, come back to this initial cursor position, and switch back to NORMAL mode.

1. Replace the first `vim` of the current line with `nop` using REPLACE mode.
2. Delete the whole line using VISUAL mode.
3. Change the first `v` of `vim` with a `w` in NORMAL mode.
4. Open a new line below and switch to INSERT mode, using only one key in NORMAL mode.
5. Insert a semi colon ":" at the end of the line.

Deleting In Vim (NORMAL mode)

We've already seen in rank I how to delete some text with the operators delete and change. Combined with themselves, a motion, or a text-object, they're the most useful tools to erase text from existence.

That said, deleting in Vim can have some surprising effects when you also want to yank and put (paste) some text.

Delete, Yank, and Put

Let's look at an example to illustrate one of the burden Vim novices might stumble upon. Let's say that you've the following text (you guessed it, from the book companion):

```
local function restorePosition()
```

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

1. Hit `yiw` to yank inside the word `local`. It ends up in the clipboard, to put it (paste it) later.
2. Hit the motion `w` to move to the `f` of `function`.
3. Hit `daw` to delete around the word `function`.
4. Hit `P` to put (paste) the text before the character under the cursor.

Here are the different steps:

Step	Keystroke	Result
1	`yaw`	`local function restorePosition()`
2	`w`	`local function restorePosition()`
3	`daw`	`local restorePosition()`
4	`P`	`local function restorePosition()`

Many would expect to put the word `local` in the fourth step. Instead, we put what we've deleted, the word `function`. In fact, we're back to our initial text.

It's because the NORMAL mode keystroke `P` (or `p`) doesn't only put back the last yanked text, but also the last deleted text.

For now, you could think of this behavior as throwing whatever you delete into your clipboard, as it was a trash bin. It means that anything you delete will always be brought back if it's the last thing you did before hitting a put keystroke (`p` or `P`) in NORMAL mode.

This behavior will make more sense when we'll look at Vim's registers in rank V. From there, we'll have the opportunity to choose what we want to put: what we've yanked, what we've deleted, and more. Until then, bear with me; it might be quite annoying at first, but it's worst the pain.

Cross the Unwanted Characters

There are two NORMAL mode keystrokes you can use to delete single characters. Here they are:

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

Both are equivalent to the NORMAL mode keystrokes `dl` and `dh` respectively, which are the combinations of the delete operator and the motions `l` (right) and `h` (left).

Help Yourself

`:help deleting`

It's Playtime!

Exercise B – Deleting in Vim

Open the file "functions.lua" from the book companion. Using the `hjkl` keys in NORMAL mode, move your cursor to the following position:

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

You should go through the steps below one after the other, keeping the changes you've made at each step.

1. Delete the word `restore`.
2. Replace the uppercase `P` of `Position` with the lowercase `p`.
3. Move to the `n` of `position`.
4. Put what you've deleted, to end up with `positionrestore`.
5. Delete the `e` of `positionrestore` with one key.

Navigating Vim Help (COMMAND-LINE Mode)

This book doesn't try to explain everything you can do in Vim; instead, it tries to highlight its most useful functionalities. In contrast, Vim help explains almost everything, making it more complicated, more dense, but also more complete.

Vim help is your most precious ally in your adventure in Vim Land: it can answer most of your questions. But to get the answer, you need first to learn how to ask properly.

Asking for Help

Here are the most useful Ex commands you can use to get the information you need from Vim help:

Ex command	Short name	Description
`:help`	`:h`	Open the main help file, including an extensive table of content.
`:help {subject}`	`:h {subject}`	Open the Vim help file about `{subject}` in a split window.

Also, as we saw in the preface, Vim help use specific notations to describe Ex commands and keystrokes. If you don't understand some of them while consulting Vim help, look at `:help notation`.

Help Yourself

`:help helphelp`
`:help helpgrep`
`:help notation`

Follow The Definition

Vim help is a set of text files. Thanks to a tag file (we'll see this concept much further in the book, in rank XVII), you can also jump to the definitions of some specific keywords. These links appear in a different color in Vim.

If you want to follow one of these keywords, place your cursor on the keyword of interest, and hit `CTRL-]`. You'll then jump to another part of the same help file, or even to another file. To jump back, you can hit `CTRL-o` (we'll come back to this specific keystroke in rank IV).

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 definition of `bars`, at the end of the same file. To come back to where you were, hit `CTRL-o`.

To summarize:

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

To close Vim help, you can run the Ex command `:q` in COMMAND-LINE mode.

What about displaying the table of content of the current help file, to make your navigation even easier? Here's the NORMAL mode keystroke you'll need:

Keystroke	Description
`gO`	Open a new window with the table of content (TOC) of the current help file. If the TOC is already open, move your cursor inside.

You can then jump to any section by moving your cursor on it and hit `<enter>`. You can close the table of content with the Ex command `:q`.

Help Yourself

```
:help CTRL-]
:help CTRL-O
```

Finding What Your Heart Desire

We've seen just above that you can search through Vim help using `:help {subject}`. But what should be this `{subject}`? It's sometimes difficult to come up with a good one in order to get the information you seek.

Thankfully, you can follow a set of rules depending on what you're searching. Here are some examples of general patterns you can use:

Ex command	Description
`:help {a-specific-topic}`	Search for `{a-specific-topic}`. A specific entry in Vim's help is often composed of a couple of words separated with dashes `-`.
`:help {keystroke}`	Search for a NORMAL mode `{keystroke}`.
`:help CTRL-{character}`	Search for a NORMAL mode keystroke `CTRL-{character}`.
`:help i_CTRL-{character}`	Search for an INSERT mode keystroke `CTRL-{character}`.
`:help c_CTRL-{character}`	Search for a COMMAND-LINE mode keystroke `CTRL-{character}`.
`:help -{option}`	Search for an `{option}` you can give to the `vim` (or `nvim`) shell command.

To drive the point home, here are more concrete examples:

Example	Description
`:help insert-mode`	Search for the topic `insert-mode`.
`:help o`	Search for the keystroke `o` in NORMAL mode.
`:help daw`	Search for the keystroke `daw` in NORMAL mode.
`:help CTRL-o`	Search for the keystroke `CTRL-o` in NORMAL mode.
`:help i_CTRL-o`	Search for the keystroke `CTRL-o` in INSERT mode.
`:help c_CTRL-f`	Search for the keystroke `CTRL-o` in COMMAND-LINE mode.
`:help CTRL-r_CTRL-r`	Search for the keystroke `CTRL-r` followed by `CTRL-r`.
`:help -u`	Search for the option `-u` you can give to `vim` (or `nvim`) shell command.

There are more patterns you can use with `:help`, to specify what you're actually searching. We'll discover them throughout the book.

It's Playtime!

Exercise C – Using Vim Help

Consider the following from Vim help:

```
:[range]m[ove] {address}
```

1. What part of the above Ex command is mandatory? What part is optional?
2. What Ex command would you use to find the above entry in Vim help?
3. What Ex command would you use to find the NORMAL mode keystroke `CTRL-v` in Vim help?
4. What Ex command would you use to find the INSERT mode keystroke `CTRL-v` in Vim help?
5. What Ex command would you use to find the COMMAND-LINE mode keystroke `CTRL-v` in Vim help?

Configuring (Neo)Vim: What Language to use? (Customization)

Once upon a time, Vimscript was created to configure Vim. It's straightforward to use for this limited job; Vimscript has useful Ex commands to customize Vim as you see fit.

But, over the years, Vimscript mutated: it became a more general scripting language. It's also where things begin to get ugly: Vimscript has many pitfalls and weird design decisions. For anything more than simple Vim configuration, it can be painful to understand, use, and debug.

It's where the developers of Neovim come into the picture. They basically took Vim's source code to make their own editor. One of the goal was to propose another language than Vimscript to configure everything, and they agreed on using Lua.

As a result, if you decide to use Neovim, you can write your configuration in Vimscript, Lua, or a mix of both.

Following Neovim's evolution, Vim also decided to propose a new and better language for its configuration: Vim9 Script, available from Vim version 9 on.

Don't be confused: what's called commonly Vimscript in this book (and all over the Internet) is not Vim9 Script. In fact, I won't speak about Vim9 Script at all in this book. Simply be aware that you can write your vimrc in Vim9 Script in Vim if you want to (but not in Neovim).

Actually, most of the configuration in this book will be in Vimscript. I'll try to provide the Lua version also, but only as a second step. In my opinion, Lua is especially useful when the configuration becomes more complex.

Why not configuring everything in Lua at the first place? For different reasons:

1. Many good resources and useful functions available on the glorious Internet are written in Vimscript. Understanding the basics of this language will help you understand and customize what you'll find online.
2. Vimscript will continue to be supported by both Vim and Neovim.
3. At the time of writing, some Vimscript functions don't have any equivalence in Lua. You'll have to use Vimscript in these cases.
4. Basic Vimscript is still used when crafting Ex commands in COMMAND-LINE mode.
5. I want this book to be useful for Vim users too, not only Neovim users.

If you use Neovim, remember that you can call some Vimscript in Lua, and vice-versa.

If you want most of your Neovim configuration in Lua, use the file `init.lua` as your vimrc. If you want most of your configuration in Vimscript (what I personally do), use `init.vim` as your vimrc.

Help Yourself

`:help vimscript`
`:help lua-guide` (only for Neovim)

Exercises

Exercise 1 – Visual Mode

Open the file "functions.lua" from the book companion. 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
```

Cancel your selection by hitting `<esc>` after each step.

1. Select the word under the cursor (without its following `<space>`).
2. Select everything until the word `and` on the same line (without including it), and yank the text selected.
3. Find a shorter NORMAL mode keystroke to do the same yank than the previous question.
4. Select everything inside the next parentheses on the same line.
5. Select the entire line and the line below.
6. Select the entire function in [visual] mode.

Exercise 2 – Help Yourself!

Using Vim 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 quickfix list?
4. Find information about the Ex command "save"?

Beyond the Rank

These exercises are more difficult. The solutions will often involve some complementary concepts not seen in this rank.

Exercise 3 – More Visual Mode

Open the file "functions.lua" from the book companion. 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
```

Cancel your selection by hitting `<esc>` after each step.

1. Select the current line and the next two lines only hitting a keystroke of two keys.
2. Select and save the current line to the filepath "/tmp/file".
3. Select the whole function, and indent your selection two times
4. Select everything from your cursor position until the first occurrence of the word `normal` on the line below.

Exercise 4 - More Help, Please

Using Vim help, how would you:

1. Find information about the option `'filetype'`?
2. Find information about the keystroke `CTRL-r CTRL-r` in COMMAND-LINE mode?
3. Find information about the `<esc>` keystroke in INSERT mode?
4. Find information about the Vimscript function `expand`?
5. Find information about the error message `E492: Not an editor command: abcde`?

Exercise 5 - Swapping

Open the file "functions.lua" from the book companion. 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
```

For each step below, undo all your changes and come back to this initial cursor position. Use NORMAL mode for each step.

1. Swap the character `l` 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`, only using five keys.
4. Swap the entire function `restorePosition` with `deleteTrailingWS` only using five keys.

Exercises – Solutions

Other solutions than the ones presented here are possible.

Exercise A – Even More Vim Modes

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

Exercise B – Deleting in Vim

Question	Keystroke	Result
start		`local function restorePosition()`
1	`dtP`	`local function Position()`
2	`rp`	`local function position()`
3	`e`	`local function positionn()`
4	`p`	`local function positionrestore()`
5	`x`	`local function positionrestor()`

Exercise C – Using Vim Help

1. `[range]` and `[ove]` are optional; `:` , `m` and `{address}` are mandatory. For example, all following Ex commands are valid:

```
:move 2
:m 2
:3mo 2
:mov 2
```

2. `:help :move`
3. `:help ctrl-v`
4. `:help i_ctrl-v`
5. `:help c_ctrl-v`

Exercise 1 – Visual Mode

1. Hit `viw` in NORMAL mode. The keystroke `vaw` selects the following space, and `vw` selects until the first letter of the next word.

2. Hit `vtay` in NORMAL mode.
   - `v` switches to VISUAL mode.
   - `ta` selects everything till the next `a` on the current line.
   - `y` yanks the selection.
3. Instead of hitting `vtay`, it's easier (and faster) to hit `yta` in NORMAL mode.
4. Hit `vi(` or `vib` (b is for block). If your cursor is not on (or in) parentheses, it will operate on the next ones on the current line.
5. Hit `Vj` in NORMAL mode.
   - `V` switches to VISUAL mode linewise.
   - `j` move your cursor to the next line, extending the selection in the process.
6. Hit `vip` in NORMAL mode to select a whole paragraph.

Exercise 2 - Help Yourself

To find information in Vim help, use the Ex command `:help`.

1. `:help replace-mode`
2. `:help *`
3. `:help quickfix` - unfortunately, `:help quickfix-list` doesn't work.
4. `:help :save` - don't forget the colon `:` if you specifically search for an Ex command.

Beyond The Basics Solutions

Exercise 3 - More Visual Mode

1. You can use a count (see rank III): hitting `3V` in NORMAL mode will select the current line as well as two more lines below.
2. Follow these steps:
   - Hit `V` to select the current line.
   - Hit `:` to switch to COMMAND-LINE mode.
   - You'll see `'<,'>` appearing in your command-line. Write `write /tmp/file` afterward, followed by `<enter>`.
   - You can run `:edit /tmp/file` to create a new buffer (see rank IV) linked to this file.
3. You can select the whole option by hitting `vip` in NORMAL mode. You can then indent it using the VISUAL mode keystroke `>`. Here are the different ways you can indent it two times:
   - Hit `>`, and then repeat the last action with `.` (see rank **TODO**).
   - Use a count: hit `2>` (see rank TODO.
4. You can hit `v/normal`. The search `/normal` acts here as a motion.

Exercise 4 - More Help, Please

1. `:help 'filetype'` - don't forget the single quotes `'` to specifically
2. `:help c_CTRL-r_CTRL-r` - it means hitting `CTRL-r` followed by `CTRL-r` in COMMAND-LINE mode
3. `:help i_<esc>`.
4. `:help expand()`
5. `:help E492` - not all Vim messages have a clear description in Vim help however.

Exercise 5 – Swapping

Question	Keystroke	Result
start		`local function restorePosition()`
1	`xp`	`olcal function restorePosition()`
2	`ddp`	`local function restorePosition()`
3	`dwelp` or `dwf<space>p`	`function local restorePosition()`
4	`dap}p`	`local function restorePosition()`

Rank III – Beginner

If you think I'm exaggerating by saying that you'll be able to write without even knowing it after this rank, you're right. Actually, you'll soon realize that I'm exaggerating in the whole book.

It's time to add more concepts to your Vim Grimoire, to cast even more powerful spells in Vim Land. More specifically, we'll see in this rank:

- How to search some text in the current file.
- How to repeat NORMAL mode keystrokes by adding a count as prefix.
- What are Vim messages, and how to display its history.
- What are Vim options, and how to change their values.

This arcane knowledge is forever useful in Vim Land. Let's take our wizard hat without waiting, you crazy sorceress.

Searching in a File

There are two ways to search in your current file: using the COMMAND-LINE mode, or using some handy keystrokes in NORMAL mode.

Vim Search in COMMAND-LINE Mode

As we saw in rank I, we can hit the keystroke `:` in NORMAL mode to switch to COMMAND-LINE mode and run Ex commands. We can also search throughout the file currently open in COMMAND-LINE mode thanks to these keystrokes:

Command	Description
`/{pattern}`	Switch to command-line mode, and Search forward for `{pattern}` from the cursor position.
`?{pattern}`	Switch to command-line mode, and search backward for `{pattern}` from the cursor position.

The `{pattern}` can be a regular expression (see rank IX). It means that regex metacharacters (like `.` for example) need to be escaped (`\.` for example) to match the literal characters.

When you hit `<enter>` after typing your `{pattern}`, you'll switch back to NORMAL mode, and you'll move directly to the first match. If there's no match, Vim will simply display an error `Pattern not found`.

If there are more than one match, you can move from the current one to the next by using these NORMAL mode keystrokes:

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

Let's look at an example in our favorite file "functions.lua" from the book companion:

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

If you hit `/` in NORMAL mode, you'll switch to COMMAND-LINE mode; your cursor will end up at the bottom of Vim. From there, you can type the pattern you want to search. For example, you can type `/vim`. Then, hitting `<enter>` will move your cursor to the first match "vim" and switch back to NORMAL mode.

You can then hit `n` to go from match to match. For example, hitting `n` two times will move your cursor on the last `vim` of the current line:

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

That's not all: searching can be a motion. It means that you can combine an operator with a search. As for any motion, the operator will operate from the cursor position to the first match.

For example, let's come back to this cursor position in our file:

```
local function restorePosition()
```

If you hit `d/ition<enter>` in NORMAL mode, you'll delete everything from the cursor position to the first match. The result:

```
local Position()
```

You can also search for regular expressions, not only for plain text. We'll look at Vim regular expressions in rank XI.

Searching the Word Under the Cursor

We can also search directly for the word under the cursor using NORMAL mode keystrokes:

Keystroke	Description
`*`	Search forward for the word under the cursor.
`g*`	Search forward for the substring under the cursor.
`#`	Search backward for the word under the cursor.
`g#`	Search backward for the substring under the cursor.

Let's look a this example:

```
vim or neovim, that is the question: whether 'tis nobler in the mind to vim
↪ the slings and arrows of neovim fortune, or to take away the uppercase
↪ against a sea or characters
```

This gibberish is also available in the book companion (what a gift), for you to try out the different keystrokes here. If you hit `*` multiple times, you'll only go through all the `vim` words. But if you use `g*`, you'll also match the substring `vim` of `neovim`.

You can also hit the normal mode keystrokes `n` and `N` after any keystroke described above, to move from one match to another.

Help Yourself

`:help search-commands`

It's Playtime!

Exercise A – 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
```

You should go through the steps below one after the other, keeping the changes you've made at each step. Use Vim search for each question.

1. Search for the word `vim` and move your cursor to the first match.
2. Move to the next match.
3. Move back to the first match.
4. Move up using the motion `k`, then move to the next match of the word `function` under the cursor.

Count: Repeating Keystrokes (NORMAL mode)

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

Most NORMAL mode motions we've seen in rank I can be repeated; we only need to add a count (a number of repeat) as prefix.

For example, let's open in Vim the file "functions.lua" from the book companion. Let's put our cursor on the following character:

```
local function restorePosition()
```

Let's add a count of "2" to the "word" motion, by hitting `2w` in NORMAL mode. The result:

```
local function restorePosition()
```

We've applied the motion "word" two times!

What about operators? If you move to the beginning of the line and then hit `2daw`, you end up

with the following:

```
restorePosition()
```

We've deleted two words.

In fact, many NORMAL mode keystrokes apply a count by default if you don't specify one; it's often "1", for the keystroke to be applied only once.

If you apply a count to both a motion and an operator, they are multiplied. For example, if you hit `2d3w`, you're effectively deleting 2 times 3 words, so 6 words.

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

That being said, you don't need to know exactly what count you need. You can try to use a count and, if it brings you closer to your goal, it's already a win. You can then finish your task using a couple more keystrokes.

The advice above stays true for almost everything you can do in NORMAL mode: if you don't manage to finish your task with a couple of keystrokes (even if you know it's possible to do so), that's fine. Try to learn from the experience and you'll get better.

Using count is also more useful when the count stays low. It's easy to predict what will happen when you want to delete a couple of words, lines, or paragraphs. Not so much if you try to delete 28 of them.

A last word about count: since many NORMAL mode keystrokes can take a count as prefix, I won't always specify when a count is possible in this book, for readability purposes. Thankfully, Vim help always specify `[count]` when you can add one to a specific keystroke.

I'll come back to this functionality throughout the book when it's the most useful, with practical examples.

Help Yourself

`:help count`

It's Playtime!

Exercise B – Count: Repeating Keystrokes

Open the file "functions.lua" from the book companion. 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
```

For each step below, undo all your changes and come back to this initial cursor position.

1. Move to the third letter `v` of the current line, as follows:

```
if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <=
 ↪  vim.fn.line("$") then
```

2. Insert two new empty lines below the current one.
3. Delete in one operation the current line, as well as the two next ones.
4. Delete in one operation until the third word `fn` of the current line.

Vim Messages (COMMAND-LINE mode)

Sometimes, Vim will speak to you; it will display some messages at the bottom, where you normally write your Ex command or searches. For example, if you search for some pattern in your current text but there is no match, Vim will tell you with verve and panache, as we saw at the beginning of this rank.

Good news everyone: nothing is eternal. The displayed messages will disappear if Vim needs to display something else there, like another message, or the Ex command you want to run.

That's unfortunate. Thankfully, if you want to bring back some old messages you might have missed, you can use the following Ex command:

Ex command	Short name	Description
`:message`	`:mes`	Display the message history.

It can also be very useful to craft your own messages, to debug your vimrc for example. To do so, you can use other Ex commands:

Ex command	Description
`:echo {expression}`	Display the output of `{expression}`, but doesn't keep it in the message history.
`:echomsg {expression}`	Display the output of `{expression}`, and keep it in the message history.
`:echoerr {expression}`	Display the output of `{expression}` as an error message, and keep it in the message history.

For example, you can try the following:

```
:echo 'hello'
:echomsg 'greetings' 'bonjour'
```

Both message will be displayed at the bottom of Vim, but only `greetings` and `bonjour` will be saved in the message history. You can even do more than displaying string of characters; you can display actual Vimscript expressions. For example:

```
:echo 2+2
:echo system('ls')
:echomsg expand('%') 3+3 'hello'
```

Help Yourself

`:help message-history`

It's Playtime!

Exercise C – The Message History

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
```

You should go through the steps below one after the other, keeping the changes you've made at each step. Use Vim search for each question.

1. Try to search for the pattern "emacs". What message Vim gives you?
2. How would you display the message history, including the message displayed in question 1?
3. How would you display the output of the shell command `man ascii` in Vim?

Vim Options (Customization)

Vim comes with many options in order to customize the editor as we see fit. They are similar to the settings you can fine-tune in many other editors. We've already seen some of them in rank I; let's now explore how they work.

Setting Options

An option is similar to a variable which can be set to a specific type of value. This type can be:

- Boolean.
- Number.
- String of characters.

The Ex command you can use to set the value of an option depends on the option's type.

Help Yourself

```
:help options
:help set-option
```

Displaying the Value of An Option

But before setting the value of an option, you'll often need to check its current value. You can do so with the following Ex command:

Ex command	Description
`:set {option}?`	Return the `{option}`'s value.

For example, the Ex command `:set clipboard?` will give you the value of the option `'clipboard'`. It's a bit like asking the option for its value.

Setting a Boolean

Options with boolean values can simply be turned on and off. You can use the following Ex commands to do so:

Ex command	Description
`:set {option}`	Set the `{option}` on.
`:set no{option}`	Set the `{option}` off.
`:set {option}!`	Toggle the `{option}`.

For example, we've turned on the option `'number'` in rank I, by adding the following to our vimrc:

Language	Config
Vimscript	`set number`
Lua	`vim.opt.number = true`

If you run the following Ex command, you'll turn it off:

Language	Ex command
Vimscript	`:set nonumber`
Lua	`:lua vim.opt.number = false`

Setting a Number

It's easy to set an option accepting a number as value:

Ex command	Description
`:set {option}={value}`	Set the `{value}` to the `{option}`.

For example, you can hide or show the status line at the bottom of Vim thanks to the option `'laststatus'`:

Ex command	Description
`:set laststatus=1`	Show the status line
`:set laststatus=0`	Hide the status line

It's important to remember that you shouldn't put spaces around the equal sign `=`. For example, the following Ex command won't work:

```
:set laststatus = 0
```

Setting a String of Characters

You can use one of the following Ex command to set an option accepting a string of characters as value:

Ex command	Description
`:set {option}={value}`	Set the `{value}` to the `{option}`.
`:set {option}+={value}`	Append a substring `{value}`, separated with a comma, to the `{option}`'s existing value.
`:set {option}-={value}`	Delete the substring `{value}` to the `{option}`'s existing value.

The two last Ex commands are useful because, for some options, you can actually give them more than one value. The value of the option itself will still be a string of characters, but divided by commas to represent the different values.

For example, in rank I, we appended a value to the option `'clipboard'` in our vimrc as follows:

Language	Config
Vimscript	`set clipboard+=unnamedplus`
Lua	`vim.opt.clipboard:append({'unnamedplus'})`

Now, let's try to run the following:

```
:set clipboard+=unnamed
:set clipboard?
```

The second Ex command should have added the value `unnamed` to the option `'clipboard'`. Congratulations! You successfully appended a new value to your option. One of your lifetime goal has just been achieved.

To delete the value `unnamedplus`, you can run the following:

```
:set clipboard-=unnamedplus
```

To reset the option to only `unnamedplus` regardless of its current value, you can run the following:

```
:set clipboard=unnamed
```

Setting Strings with Spaces

When setting a string to an option, you'll have to escape white spaces using backslashes `\`. For example:

```
:set formatprg=prettier\ --stdin-filepath\ %
```

That's annoying really fast. Thankfully, we can also use the following Ex command to set a string as value without the need to escape the whitespaces:

Ex command	Description
`:let &{option}='{value}'`	Set the `{value}` to the `{option}`.

For example, the two following Ex commands are equivalent:

```
:set formatprg=prettier\ --stdin-filepath\ %
:let &formatprg='prettier --stdin-filepath %'
```

Fortunately, you won't have this problem in Lua:

```
:lua vim.opt.formatprg='prettier --stdin-filepath %'
```

We'll see what the option `'formatprg'` stands for in rank VII

Help Yourself

`:help :let-option`

Setting an Option to its Default Value

It's easy to fiddle with an option and change its value, but what if you want to set the option back to its default? You can use the following Ex command to do so:

Ex command	Description
`:set {option}&`	Reset the `{option}` to its default value.

For example, if you want to set the option `'laststatus'` back to its default value, you can run the following Ex command:

```
:set laststatus&
```

This is really useful when you begin to experiment with different options by changing their values.

Setting Options Interactively

There's another way to set Vim options:

Ex command	Description
`:options`	Open a new window to display and set Vim options.

You can then go through the list of options. If you hit `<enter>` on the option itself, you'll open Vim help. If you hit `<enter>` on the value of the option, you'll be able to change it.

This Ex command can also be useful to discover new options; they're even grouped by functionality!

That being said, the sheer number of options here can be daunting; don't try to set all of them at once, or you'll go crazy by so many possibilities. I don't want this book to make you insane.

Persisting an Option's Value

As we already saw, if you want an option to always have the same customized value when you open Vim, you need to set it in your vimrc. It's an important concept, that's why I'm repeating myself a bit here.

Also remember: you don't need the prefix `:` if you want to put some Ex commands in your vimrc, like setting some options. Instead of writing `:set laststatus=0`, you can write `set laststatus=0`. It's a convention Vim users follow.

Searching an Option in Vim Help

All of that is great, but an important question remains:

How do we know what value we can give to a specific option? For example, how do we know that setting "1" to the option `'laststatus'` will show the status line?

You can get this information (and much more) in Vim help. To search for an option specifically, you'll have to surround the name of your option with single quotes. For example:

```
:help 'number'
:help 'laststatus'
```

That's also why I surround options with single quotes in this book (i.e `'laststatus'`). It's to remind you of this important tip.

Some Useful Options

To conclude this section, let's look at a couple of useful options. They all accept a boolean as value:

Option	When "on"	Default
`'ignorecase'`	Search ignore the case sensitivity (uppercase and lowercase characters are equivalent).	off
`'smartcase'`	Search is case sensitive if there is one or more uppercase in the pattern. Needs the option 'ignorecase' to be on.	off
`'hlsearch'`	Highlight the matching search pattern. Use the Ex command `:nohlsearch` (or `:noh`) to turn the highlight off.	off (Vim) on (Neovim)
`'autowriteall'`	Automatically write open files.	off
`'incsearch'`	Display the matches in the current buffer while searching.	on

The options `'ignorecase'`, `'hlsearch'`, and `'smartcase'` will be turned "on" in the vimrc of the book companion, but feel free to do as you wish in your own vimrc.

Lastly, options have also short names, like Ex commands. For example, you can switch on the option `'ignorecase'` with `:set ignorecase` or `:set ic`. That said, I'd encourage you to always write the long name of options in your vimrc, for a better readability.

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

Help Yourself

`:help option-list`

It's Playtime!

Exercise D – Vim Options

How would you:

1. Get the value of the option `'filetype'`?
2. Toggle the boolean value of the option `'number'`?
3. Turn off the option `'compatible'`?
4. Append the substring "S" to the existing value of the option `'shortmess'`?
5. Delete the substring "S" to the existing value of the option `'shortmess'`?
6. Set back the option `'shortmess'`shortmess to its default value?
7. How would you find the short name of the option `'clipboard'`?

Exercises

Exercise 1 – Search Highlighting

Open the file "functions.lua" from the book companion.

You should go through the steps below one after the other, keeping the changes you've made at each step.

1. Search for the word "vim".
2. The matching pattern `vim` should now be highlighted. Turn off the highlighting.
3. If you search for another word, do you think it will be highlighted?
4. Disable the highlighting for all future search.

Exercise 2 – More Vim Search

Open the file "functions.lua" from the book companion. 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
```

You should go through the steps below one after the other, keeping the changes you've made at each step. Use Vim search for each step.

1. Move your cursor to this position:

```
if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= 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.

Exercise 3 – Vim Count

Open Vim in a new buffer.

You should go through the steps below one after the other. Use a count for each of them, and keep the changes you've made at each step.

1. Insert four times the word “vim” separated with spaces, and come back to the beginning of the line.
2. Select the 3 next words with only the spaces between the words. Cancel the selection.
3. Yank the current line and copy it three times.
4. Create four new lines: insert `*` at the beginning of each odd line, and let the even lines blank.

Beyond the Rank

These exercises are more difficult. The solutions will often involve some complementary concepts not seen in this rank.

Exercise 4 – Vim Messages

Open the file “functions.lua” from the book companion. 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
```

You should go through the steps below one after the other, keeping the changes you’ve made at each step.

1. Display the output of the shell command `file` with the current file as argument.
2. After pressing `<enter>` to hide the display of the previous question, display it again.
3. Clear all messages from the message history.
4. Display the last error message only.
5. Write an entry in the message history including the full path of the current file, its filetype, the current line number, and the current word under the cursor.

Exercise 5 – Setting Vim Options

1. How would you turn off the option `'expandtab'` only for the current buffer?
2. How would you verify that the option `'shortmess'` has a local value?
3. How would you easily set the option `'listchars'` to the value `tab:  ,trail:·` , but only locally to the current window?
4. How would you search in Vim help for the option `'t_SI'`?
5. How would you easily set the option `'t_SI'` to `\e[6 q` , and `'t_EI'` to `\e[2 q` ? What happens?

Exercises – Solutions

Exercise A – Vim Search

Question	Keystroke	Result
start		`#!/usr/bin/env lua`
1	`/vim<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	`k*`	`local [f]{.mne}unction deleteTrailingWS()`

Exercise B – Count: Repeating Keystrokes

Question	Keystroke	Result
start		`if vim.fn.line("'\"") > 1`
1	`3fv`	`<= vim.fn.line("$") then`
2	`2o<esc>`	
3	`3dd`	`end`
4	`d3/fn`	`fn.line("$") then`

Exercise C – The Message History

1. After hitting `/emacs<enter>`, Vim will display an error message `E486: Pattern not found: emacs`.
2. The Ex command `:message` will give you the message history.
3.
4. `:echo system('man ascii')`

Exercise D – Vim's 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&`
7. Look at Vim help with `:help 'clipboard'`. Its short name is `'cb'`.

Exercise 1 – Search Highlighting

1. Execute `/vim`.
2. Execute `:nohlsearch` or `:nohl`.
3. Yes. Searching after running `:nohl` will switch back the search highlighting.
4. Execute `:set nohlsearch` or `:set hlsearch!`.

Exercise 2 – More Vim Search

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

Exercise 3 – Vim Count

1. Hit `4ivim <esc>^` in NORMAL mode.
2. Hit `v5iw<esc>` in NORMAL mode - each space count as a word here.
3. Hit `yy3p` in NORMAL MODE
4. Hit `2o*<enter><esc>` in NORMAL mode.

Exercise 4 – Vim Messages

1. `:echo system('file ' . expand('%'))`
2. The output of external commands is not saved in the message history. But can hit `g<` in NORMAL mode to display back the last output, wherever it comes from.
3. `:messages clear`
4. `:echo v:errmsg`
5. `:echomsg expand('%:p') &l:filetype line('.') expand('<cword>')`

Exercise 5 – Setting Vim Options

1. `:setlocal noexpandtab`
2. `:help 'shortmess'`. The option is only global, it can't be set locally.
3. `:let &l:listchars='tab:> ,trail:-,nbsp:+'`
4. In Vim, you can do the usual `:help 't_SI'`. Surprisingly, it doesn't work in Neovim. You need to do `:help t_SI` there.
5. You can set these two options as follows:

```
let &t_SI = "\e[6 q"
let &t_EI = "\e[2 q"
```

It will normally give you a cursor block in NORMAL mode, and a cursor line in INSERT mode.