

Learning to play Vim



Master the best text editor,
from Beginner to Composer

Matthieu Cneude

Learning to Play Vim

Matthieu Cneude

Contents

Acknowledgments	11
Preface	12
Vim Or Neovim? That's the Question	12
What This Book is not About	13
What You Need to Follow Along	13
How To Get the Most Out of this Book	14
Structure of the Book	14
Notation Conventions	15
Playing Vim: The Exercises	16
Becoming a Vim Player	17
Vim is an Instrument	17
The Power is in Your Fingers	17
Efficient Typing: the Two Rules	18
The First Week	18
The Second Week	19
Speed and Accuracy	19
Keyboard Layouts	19
Practice, Practice, Practice	19
Rank 01 - Rookie	20
A Modal Editor	20
The NORMAL Mode	20
The INSERT Mode	21
The COMMAND-LINE Mode	22
Moving Around with Motions (NORMAL mode)	24
Ditching the Arrow Keys	24
Horizontal Motions	25
Beginning, Middle, and End of Line	26
Vertical Motions	27
Undo and Redo (NORMAL mode)	28
Operators, Motions, and Text-Objects (NORMAL Mode)	29
The Operators	29
Operators and Motions	29
Operators and Text-Objects	30
Bending Vim to Your Will (Customization)	32
The Main Configuration File: the vimrc	32
First Configuration	33
Clipboard Management	34
Improving Vim's Defaults	35
The Configuration Addiction	36
Debugging Your Configuration	36
Exercises	38
Beyond the Rank	39
Exercises - Solutions	42

Rank 02 - Novice	46
Even More Vim Modes	46
The VISUAL Mode	46
The REPLACE Mode	48
More Keystrokes to Switch to INSERT Mode	49
Deleting In Vim (NORMAL mode)	50
Delete, Yank, and Put	50
Cross the Unwanted Characters	51
Navigating Vim Help (COMMAND-LINE Mode)	52
Asking for Help	52
Follow The Definition	53
Finding What Your Heart Desire	54
Configuring (Neo)Vim: What Language to use? (Customization)	55
Exercises	57
Beyond the Rank	57
Exercises - Solutions	59
Beyond The Basics Solutions	60
 Rank 03 - Beginner	 62
Searching in a File	62
Vim Search in COMMAND-LINE Mode	62
Searching the Word Under the Cursor	63
Count: Repeating Keystrokes (NORMAL mode)	65
Vim Messages (COMMAND-LINE mode)	67
Vim Options (Customization)	69
Displaying the Value of An Option	69
Setting Options	70
Persisting an Option's Value	73
Searching an Option in Vim Help	74
Some Useful Options	74
Exercises	76
Beyond the Rank	77
Exercises - Solutions	78
 Rank 04 - Aspirant	 81
Vim's Space: Buffers, Windows, and Tabs	81
Vim Buffers	81
Vim Windows	88
Vim Tab Pages	91
Scrolling (NORMAL mode)	94
Ranges for Ex Commands (COMMAND-LINE Mode)	94
The Line Specifiers	95
Ranges and VISUAL Mode Selection	96
Creating Your Own Mappings (Customization)	97
Creating Mappings for Different Modes	97
Nested and Recursive Mapping	98
Deleting a Mapping	99
Mapping and Key Notation	100
The Leader Key	101
Ambiguous Mapping	103
Some First Mappings	103
Listing Custom Mappings	104
Exercises	106
Beyond the Rank	107
Exercises - Solutions	110
Exercise Solutions	110
 Rank 05 - Intermediate	 114
Vim Registers	114

How to Use the Registers	114
The Different Types of Registers	116
Using Registers in Practice	120
Jumping Around (NORMAL mode)	121
The Jump List	121
The Change List	122
Jumping to Matching Elements	122
Jumping Paragraphs	123
Jumping Methods	123
The Command Line History (COMMAND-LINE mode)	125
Extending Vim with Plugins (Customization)	126
A Plugin to Manage Plugin	127
Installing New Plugins	127
Exercises	129
Beyond the Rank	130
Exercises - Solutions	132
Interlude 01 - Useful Vim Plugins	136
Neovim and External Dependencies	136
Plugins Extending Vim's Functionalities	136
Tree View of Your Filesystem	137
Completion, Auto-Completion, and Jump to Definition	137
Snippets	138
The Status Line	139
Vim Color Scheme	139
Rank 06 - Competent	140
Repeating Changes	140
Single Repeat	140
Complex Repeat: Recording a Macro	142
Editing in INSERT Mode	148
Moving the Cursor From Line to Line	148
Inserting and Deleting	149
Copying Characters From Adjacent Lines	149
Indenting	149
The NORMAL mode in INSERT mode	150
Editing in COMMAND-LINE mode	151
Copying From a Buffer to the COMMAND-LINE	151
Multiple Ex Commands on One Line	151
Mapping Special Arguments (Customization)	153
Using Ex commands in Mapping	154
Silencing a Mapping	154
Exercises	157
Beyond the Rank	158
Exercises - Solutions	160
Rank 07 - Proficient	165
Navigating a Project	165
The Current Working Directory	165
Changing the Global Working Directory	166
The Local Working Directories	167
Searching for a File	167
Formatting Text (NORMAL mode)	172
Internal Formatting Functionalities	172
Formatting With an External Program	176
Search and Replace With Substitute (COMMAND-LINE mode)	179
How to Substitute your Text	179
The Substitute Flags	181
Using Different Separators	182

The Scopes of Mappings and Options (COMMAND-LINE mode)	183
Option Scopes	183
Global and Local Options in Your vimrc	184
Local Mappings for Buffers	186
Exercises	189
Beyond the Rank	190
Exercises - Solutions	193
Rank 08 - Seasoned	197
Searching in Multiple Files	197
Vim Internal Search: vimgrep	197
Using an External Program	200
Indenting Your Text (NORMAL mode)	202
Keystrokes for Indenting	202
Controlling the Indentations	203
Displaying Invisible Characters	204
Indenting With an External Program	205
Charwise, Linewise, or Blockwise? (VISUAL Mode)	206
The Types of VISUAL Mode	206
Visual Mode Charwise and Linewise	207
Visual Mode Blockwise	208
Advanced Search in a Buffer (Command-Line Mode)	210
Search With Case Sensitivity	210
Displaying the Number of Results	211
Find and Replace One Occurrence at a Time	211
Repeating the Last Search	212
Search Highlighting	213
Writing Custom Functions (Customization)	214
Vimscript Functions	214
Lua Functions	219
Exercises	224
Beyond the Rank	225
Exercises - Solutions	227
Rank 09 - Adept	234
The Quickfix Lists and Location Lists	234
The Quickfix Window	234
The Location List	235
Creating Quickfix Lists	235
Creating Location Lists	236
Navigating the Current Quickfix List	236
Creating Quickfix Lists Using Vimscript Expressions	237
Valid Entries for Quickfix Lists	238
Valid Entry When Grepping	240
Executing Ex Commands on Quickfix List Entries	240
Opening Old Quickfix Lists	241
Filtering the Current Quickfix List	243
Changing Case (NORMAL mode)	244
The Shell Power in Vim (COMMAND-LINE mode)	246
Executing Shell Commands	246
Inserting Shell Command Output in the Current Buffer	248
Filtering Your Buffer Using a Shell Command	248
Using Vim Bars With Shell Commands	250
The Ex Command Execute (Customization)	251
Using the Ex Command Execute	251
Exercises	254
Beyond the Rank	255
Exercises - Solutions	257

Interlude 02 - More Useful Vim Plugins	263
A Fuzzy Finder to Find Them All	263
Git Integration	264
Linters for Vim	264
Manipulating Buffers and Windows	264
Closing Buffers Without Closing Windows	264
A New Mode to Manage Windows	265
Extending Vim Motions	265
Displaying an Outline	265
A Debugger in Vim	266
 Rank 10 - Believer	 267
Vim Regular Expressions	267
The Basics of Regular Expressions	268
Vim Magical Regexes	269
The Concept of Atom	271
Word Boundaries	272
Greedy and Non-Greedy Quantifiers	272
Vim Lookaround Assertions	274
Matching a Regex On Multiple Lines	276
Matching a Regex in a Visual Selection	277
Increasing or Decreasing Numbers (NORMAL mode)	279
Copying and Moving Text (COMMAND-LINE mode)	281
Autocommands (Customization)	283
Creating and Deleting Autocommands	283
Listing Autocommands	284
Autocommands Events	285
Multiple Events and Patterns	286
Autocommand Groups	286
Buffer-Local Autocommands	289
Autocommands in Lua	289
Exercises	292
Beyond the Rank	295
Exercises - Solutions	298
 Rank 11 - Veteran	 308
Vim Marks	308
Setting Local and File Marks	308
Listing and Deleting Marks	309
Marks in Action	310
Special Marks	311
Absolute and Relative Line Numbers (NORMAL mode)	313
Switching Between Absolute and Relative Line Number	315
The Global Ex Commands (COMMAND-LINE mode)	316
The Basics of the Global Ex Commands	316
The Global Ex Command in Action	318
Global Ex Commands and Substitutions	319
User Commands (Customization)	320
Creating User Commands	320
Listing User Commands	321
Attributes for User Commands	321
Exercises	329
Beyond the Rank	331
Exercises - Solutions	334
 Rank 12 - Expert	 350
Undo in Depth: Coming Back to Any Change	350
The Undo Tree	350
Persisting The Undo Tree	357

Creating Undo Nodes	358
Vim Abbreviations (INSERT mode)	361
Creating and Deleting Abbreviations	361
Abbreviations in Practice	362
Listing Abbreviations	364
Local Abbreviations for Buffers	365
Executing Some Vimscript When Expanding Abbreviations	365
Abbreviations and Mapping	366
The Normal Ex Command (COMMAND-LINE mode)	367
The Global Ex Commands With NORMAL Mode Keystrokes	367
The Normal Ex Command and Special Keys	368
Operating on Custom Motions or Text-Objects (Customization)	369
The OPERATOR-PENDING Mode	370
OPERATOR-PENDING Mode Mappings	370
Creating a Motion	371
Creating a Text-Object	371
Motions, Text-Objects, and Visual Mode	373
Exercises	376
Beyond the Rank	379
Exercises - Solutions	382
Rank 13 - Champion	398
Folding in Vim	398
Managing Folds Manually	398
Creating Fold Automatically	402
Motions to Move Through Folds	409
Digraphs: Inserting Special Characters (INSERT mode)	410
Listing Special Characters, Digraphs, and Unicode Numbers	411
Inserting Special Characters	411
Creating Digraphs	412
Displaying Representations of the Character Under the Cursor	413
Finding the Digraph You Need	413
Special Keywords for Ex Commands (COMMAND-LINE mode)	414
Keywords for Filepaths	414
Keywords Expanding to the Text Under the Cursor	415
Filepath Modifiers	415
Expanding Keywords in Vimscript Expression	416
The viminfo or ShaDa Files (Customization)	418
The Files	419
Defining the Information Saved	419
Exercises	422
Beyond the Basics	423
Exercises - Solutions	427
Interlude 03 - Vim Runtime Paths	450
Vim's Startup	450
Loading Order at Startup	450
Debugging Vim Startup	451
The Runtime Paths	451
Listing All Runtime Paths	451
The Default Runtime Path	452
The User Runtime Path	453
The Subdirectories of the Runtime Paths	454
Loading Scripts for Specific Filetypes (ftplugin)	454
Filetype Detection (ftdetect)	455
Loading Functions On Demand (autoload)	456
Loading Syntax Highlighting (syntax)	458
Loading Color Schemes (colors)	459

Compiling and Linting (compiler)	459
Overwriting Defaults: the After Runtime Path	460
Rank 14 - Master	462
Running External Applications for Specific Buffers	462
Choosing an External Application	462
Running an External Application	464
Parsing Error Messages	465
A Compiler Script To Lint Lua	466
Completion in Vim (INSERT mode)	470
The Completion Submode	470
Managing the Differences Between Buffers (NORMAL mode)	471
Beginning a Diff Between Windows	471
Enabling Diff	472
The Diff Folding Method	473
Jumping From Change to Change	473
Moving Changes Between Diff Windows	474
Diff in Action	474
Configuring Diffs with An Option	477
Customizing Diff Even Further	479
Debugging Your Vim Config (Customization)	480
Vim Verbosity	481
The DEBUG Mode	483
Exercises	487
Basics	487
Beyond the Basics	487
Rank 15 - Grand Master	490
Syntax Highlighting	490
Enabling and Disabling Syntax Highlighting	490
Number of Colors in your Terminal	491
Color Schemes	491
Syntax Script of the Current Buffer	492
Syntax Items and Groups	493
Highlight Groups	495
The Verbose Command	500
Syntax Highlighting and Colorscheme in Action	500
Limits of Vim Syntax Highlighting	500
Advanced Macro (NORMAL & VISUAL mode)	502
Visual Mode Macro	502
Creating Mapping from Macros	504
Recursive Macro	505
The Arglist (COMMAND-LINE mode)	507
Practical Use: Find and Replace in Multiple Files	508
Setting the Status Line (Customization)	509
Status Line Options	509
A Concrete Example	511
Status Line Separator	512
Setting your Tab Line	513
Exercises	516
Basics	516
Beyond the Basics	516
Solutions	517
Rank 16 - Hero	518
Vim's Spelling	518
Basics	518
Adding Words to Spell Files	519
Fixing Spelling with Word Suggestions	521

Navigating Through Your Wrong Words	522
The Omni-completion	523
Redirections (COMMAND-LINE mode)	523
Improving Vim Performances	525
General Profiling	525
Startup Profiling	525
Profiling Syntax Files	526
Exercises	527
Basics	527
Beyond the Basics	527
Spell	527
Exercise xxxx - Advanced Macro	527
Solutions	527
Rank 17 - Godlike	528
Jumping to Definition	528
Vim and ctags	528
Completion with tags	530
Include-Search	531
Include-search completion	535
Saving Settings and Vim Sessions (COMMAND-LINE mode)	536
Saving Vim's Options and Mapping in a File	536
Creating and Loading a Session	537
Fine Tuning Vim's Sessions	538
Managing Plugins in Vanilla Vim (Customization)	538
The Vim Native Package Manager	539
Installing New Plugins	540
Loading Plugins on Demand	541
Exercises	542
Basics	542
Beyond the Basics	542
Solutions	542
The File Manager netrw	543
Opening netrw	543
Browsing	546
Basics	546
Display	546
Filtering Display	547
Listing the Browsing History	547
Marking Files And Directories	548
Managing Files and Directories	549
Creating and Deleting	549
Renaming Files or Directory	549
Copying Files	550
Moving Files	550
File Permissions	550
Opening Files with External Applications	550
Bookmarking	551
Remote Operations and Protocols	551
Using scp via SSH	552
Using HTTP (read only)	553
Listing Directories	553
Obtaining a file	553
FTP	553
The NETRC file	554
Customizing Mapping	554
Overwriting Variables and Functions	555

Command Line Editing	555
TODO	555
Rank 18 - Composer	556
Restoring the Cursor Position	556
Loading Specific Configuration Per Project	557
Opening a Window Fullscreen	557
Output Redirection into a Scratch Buffer	561
Git Information From the Current Line	567
Displaying in the Command Line Window	567
Creating a Popup	568
CONCLUSION	569
 Annexes	 570

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 for 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 01). 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 02. It uses specific notations for different ideas, and this book use most of them:

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

Let's look at some examples:

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

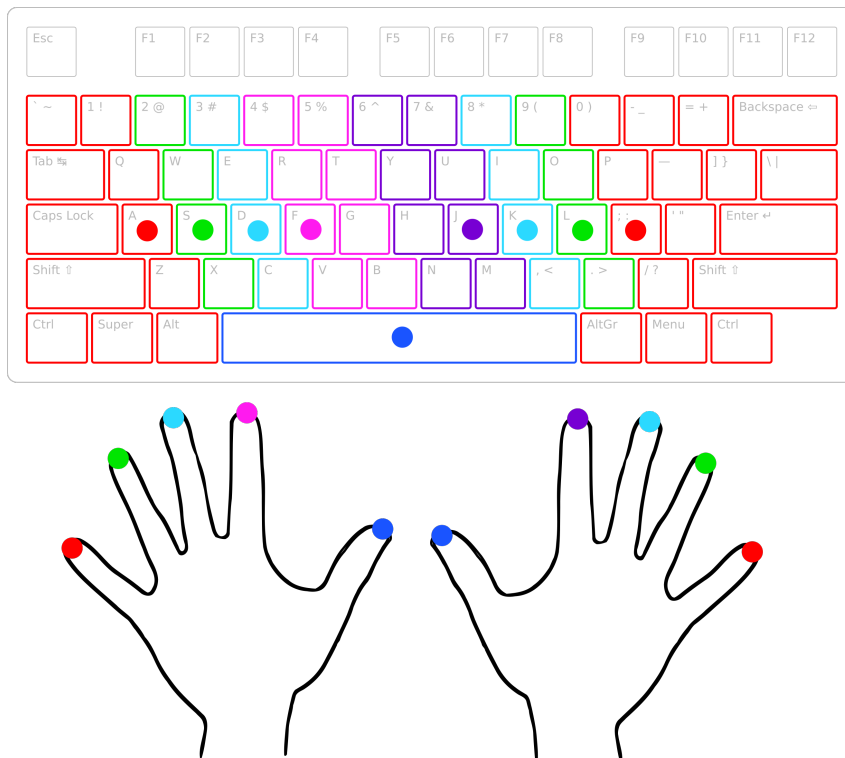
You can practice the techniques we'll see next while reading the rest of the book. It will be difficult at the beginning, but stick to it as much as you can. You won't regret it.

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 01 – 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.

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 consequences. 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.

Here are the three most useful modes:

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.

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 ()` 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
<code>i</code>	Switch from NORMAL mode to <u>I</u> NSERT mode.
<code>a</code>	Switch from NORMAL mode to INSERT mode <u>a</u> fter the character under the cursor.
<code><esc></code>	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 02**; 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
<code>:write</code>	<code>:w</code>	To <u>w</u> rite (save) the current open file.
<code>:edit {filepath}</code>	<code>:e {f}</code>	To <u>e</u> dit the file located at <code>{filepath}</code> .
<code>:quit</code>	<code>:q</code>	To <u>q</u> uit the current window.
<code>:quit!</code>	<code>:q!</code>	To <u>q</u> uit the current window without saving!

To quit Vim, you need to quit all windows. We'll see more about windows in [rank 03](#).

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
<code>:help {subject}</code>	<code>:h {subject}</code>	Open Vim's <u>h</u> elp about <code>{subject}</code> .

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 `hjk l` , 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 `h j k l` 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 `h j k l`, 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 `h j k l`. 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
<code>w</code>	Move forward to the beginning of the next <u>w</u> ord.
<code>W</code>	Move forward to the beginning of the next <u>W</u> ORD
<code>e</code>	Move forward to the <u>e</u> nd of the next word.
<code>E</code>	Move forward to the <u>e</u> nd of the next WORD.
<code>b</code>	Move <u>b</u> ackward to the beginning of the word.
<code>B</code>	Move <u>b</u> ackward to the beginning of the WORD.
<code>ge</code>	Move backward to the <u>e</u> nd 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
<code>f{character}</code>	To <u>f</u> ind a <code>{character}</code> after your cursor.
<code>F{character}</code>	To <u>f</u> ind a <code>{character}</code> before your cursor.
<code>t{character}</code>	Move <u>t</u> ill a <code>{character}</code> after your cursor.
<code>T{character}</code>	Move <u>t</u> ill a <code>{character}</code> before your cursor.

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

Keystroke	Description
<code>;</code>	Move forward.
<code>,</code>	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
<code>0</code>	To move to the first character of the current line.
<code>\$</code>	To move to the last character of the current line.
<code>^</code>	To move to the first non-whitespace character of the current line.
<code>gM</code>	To <u>g</u> o to the <u>m</u> iddle 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 <u>u</u> pward half a screen.
CTRL-d	Move <u>d</u> ownward half a screen.
CTRL-b	Move <u>b</u> ackward (upward) an entire screen.
CTRL-f	Move <u>f</u> orward (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 <u>h</u> ighest line) of the screen.
M	Move to the line at the <u>m</u> iddle of the screen.
L	Move to the <u>l</u> ast 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 `h j k l` 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 <u>u</u> ndo the last edit.
CTRL-r	To <u>r</u> edo 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 11](#).



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
<code>d</code>	To <u>d</u> elete some text.
<code>c</code>	To <u>c</u> hange some text.
<code>y</code>	To <u>y</u> ank (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
<code>dd</code>	To <u>d</u> elete the current line.
<code>cc</code>	To <u>c</u> hange the current line.
<code>yy</code>	To <u>y</u> ank 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 <u>d</u> delete the text from your cursor to the end of line. Equivalent to the alias <code>D</code> .
c\$	To <u>c</u> hange the text from your cursor to the end of line. Equivalent to the alias <code>C</code> .
y\$	To <u>y</u> ank the text from your cursor to the end of line. Equivalent to the alias <code>Y</code> (only in Neovim by default).
cw	To <u>c</u> hange from your cursor to the end of the <u>w</u> ord.
yG	To <u>y</u> ank (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 <u>p</u> ut (<u>p</u> aste) the last yanked or deleted text <i>after</i> the character under your cursor. If you yank or delete an entire line, <code>p</code> will put it after the current line.
P	To <u>p</u> ut (<u>p</u> aste) the last yanked or deleted text <i>before</i> the character under your cursor. If you yank or delete an entire line, <code>P</code> 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
<code>diw</code>	To <u>d</u> elete <u>i</u> nside the <u>w</u> ord. It deletes the current word under the cursor.
<code>daw</code>	To <u>d</u> elete <u>a</u> round the <u>w</u> ord. It deletes the current word under the cursor, as well as the whitespace following it.
<code>ciw</code>	To <u>c</u> hange <u>i</u> nside the <u>w</u> ord. It deletes the current word under the cursor and switch to INSERT mode.
<code>dip</code>	To <u>d</u> elete <u>i</u> nside the <u>p</u> aragraph.

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

In Vim 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	<code>~/.vim/vimrc</code>	Vimscript
Neovim	<code>\$XDG_CONFIG_HOME/nvim/init.vim</code>	Vimscript
Neovim	<code>\$XDG_CONFIG_HOME/nvim/init.lua</code>	Lua

When Vim starts, it reads the configuration file “`vimrc`” (or “`init.vim`”, or “`init.lua`”), and each

line is executed. To make things simpler I'll use the umbrella term "vimrc" throughout this book to speak about Vim's main configuration file, whatever its filename.

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 02](#). 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 03](#). 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:

```
-- 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`:

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

In 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 04](#).



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 02. We're also looking at the option 'hidden' more thoroughly in [rank 04](#).

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
<code>vim -u NONE</code>	Launch Vim without your vimrc.
<code>vim --noplugin</code>	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 into Vim plugins in [rank 05](#).

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 `h j k l` 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 `h j k l` 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 `h j k l` 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 `h j k l` 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		
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	locf function 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, dele`te` from the cursor position to the beginning of the next word `function`, and `diw` dele`te` 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 delate 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		<code>local function restorePosition()</code>
1	<code>dl</code>	<code>local function restorePosition()</code>
2	<code>dh</code>	<code>local function restorePosition()</code>

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		<code>local function restorePosition()</code>
1	<code>+</code>	<code>if vim.fn.line("'\"") > 1</code>
2	<code>-</code>	<code>local function restorePosition()</code>

Rank 02 – 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
<code>v</code>	Switch to <code>VISUAL</code> mode "charwise".
<code>V</code>	Switch to <code>VISUAL</code> 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
<code>vaw</code>	Select <u>v</u> isually <u>a</u> round a <u>w</u> ord.
<code>vawd</code>	Select <u>v</u> isually <u>a</u> round a <u>w</u> ord, delete it, and come back to NORMAL mode. Equivalent to the NORMAL mode keystroke <code>daw</code> .
<code>vw</code>	Select <u>v</u> isually from the cursor position to the end of the current <u>w</u> ord.
<code>v\$</code>	Select <u>v</u> isually from the cursor position to the end of line.
<code>vf,y</code>	Select <u>v</u> isually from the cursor position to the first comma, and then yank the selection.
<code>vawyp</code>	Select <u>v</u> isually <u>a</u> round a <u>w</u> ord, <u>y</u> ank it, and <u>p</u> ut (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
<code>Vy</code>	Select the current line and yank it. Similar to the NORMAL mode keystroke <code>yy</code> .
<code>Vj</code>	Select the current line and the line below.
<code>Vc</code>	Select the current line and <u>c</u> hange it. Equivalent to the NORMAL mode keystroke <code>cc</code> .
<code>VGd</code>	Select every line from the current one until the last, and delete the text selected.
<code>Vyp</code>	Select the current line, yank it, and <u>p</u> ut (paste) it below.

There is also the VISUAL mode per block (“blockwise”), but we’ll see that later in the book, in rank 08.

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 01](#)). 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 <u>R</u> EPPLACE 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
-----	-----
<code>`r`</code>	Replace the character under the cursor.
<code>`R`</code>	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 01](#), 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 <u>i</u> nsert before the current character.
I	To <u>i</u> nsert before the first non-blank character of the current line.
a	To insert <u>a</u> fter the current character.
A	To insert <u>a</u> fter the end of the current line.
o	To <u>o</u> pen a new line below the current one and switch to INSERT mode.
O	To <u>o</u> pen a new line above the current one and switch to INSERT mode.
<esc>	Switch back to NORMAL mode.
CTRL-c	
CTRL-[



Help Yourself

```
:help insert-mode
```

```
:help Q_in
```



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 01](#) 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 05](#). 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
<code>x</code>	Delete the character under the cursor.
<code>X</code>	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
<code>:help</code>	<code>:h</code>	Open the main help file, including an extensive table of content.
<code>:help {subject}</code>	<code>:h {s}</code>	Open the Vim help file about <code>{subject}</code> 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 17](#)), 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 04](#)).

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
<code>CTRL-]</code>	Jump to the definition of the keyword under the cursor.
<code>CTRL-o</code>	Jump back to your <u>older</u> 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
<code>g0</code>	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
<code>:help {topic}</code>	Search for a specific <code>{topic}</code> . A <code>{topic}</code> in Vim's help is often composed of a couple of words separated with dashes - .
<code>:help {keystroke}</code>	Search for a NORMAL mode <code>{keystroke}</code> .
<code>:help CTRL-<code>{character}</code></code>	Search for a NORMAL mode keystroke <code>CTRL-<code>{character}</code></code> .
<code>:help i_CTRL-<code>{character}</code></code>	Search for an INSERT mode keystroke <code>CTRL-<code>{character}</code></code> .
<code>:help c_CTRL-<code>{character}</code></code>	Search for a COMMAND-LINE mode keystroke <code>CTRL-<code>{character}</code></code> .
<code>:help -<code>{option}</code></code>	Search for an <code>{option}</code> you can give to the shell command <code>vim</code> (or <code>nvim</code>).

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

Example	Description
<code>:help insert-mode</code>	Search for the topic <code>insert-mode</code> .
<code>:help o</code>	Search for the keystroke <code>o</code> in NORMAL mode.
<code>:help daw</code>	Search for the keystroke <code>daw</code> in NORMAL mode.
<code>:help CTRL-o</code>	Search for the keystroke <code>CTRL-o</code> in NORMAL mode.
<code>:help i_CTRL-o</code>	Search for the keystroke <code>CTRL-o</code> in INSERT mode.
<code>:help c_CTRL-f</code>	Search for the keystroke <code>CTRL-o</code> in COMMAND-LINE mode.
<code>:help CTRL-r_CTRL-r</code>	Search for the keystroke <code>CTRL-r</code> followed by <code>CTRL-r</code> .
<code>:help -u</code>	Search for the option <code>-u</code> you can give to <code>vim</code> (or <code>nvim</code>) 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		<code>if vim.fn.line("'\"")</code>
1	<code>Rnop</code>	<code>if nop.fn.line("'\"")</code>
2	<code>Vd</code>	<code>vim.cmd([[normal g]])</code>
3	<code>rw</code>	<code>if wim.fn.line("'\"")</code>
4	<code>o</code>	<code> </code>
5	<code>A:</code>	<code><= vim.fn.line("\$") then:</code>

Exercise B – Deleting in Vim

Question	Keystroke	Result
start		<code>local function restorePosition()</code>
1	<code>dtP</code>	<code>local function Position()</code>
2	<code>rp</code>	<code>local function position()</code>
3	<code>e</code>	<code>local function position()</code>
4	<code>p</code>	<code>local function positionrestore()</code>
5	<code>x</code>	<code>local function positionrestor()</code>

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 03](#)): 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 04](#)) 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		<code>local function restorePosition()</code>
1	<code>xp</code>	<code>olocal function restorePosition()</code>
2	<code>ddp</code>	<code>local function restorePosition()</code>
3	<code>dwelp</code> or <code>dwf<space>p</code>	<code>function local restorePosition()</code>
4	<code>dap}p</code>	<code>local function restorePosition()</code>