Learning to play Vim



Master the best text editor,
from Beginner to Composer

Matthieu Cneude

Learning to Play Vim

Matthieu Cneude

Learning to Play Vim

Copyright © 2026 Matthieu Cneude

All rights reserved.

While every precaution has been taken in the preparation of this book, the author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Contents

Acknowledgments **i**

Preface **ii**
Vim Is Not a Word Processor . . . ii
What to Use: Vim or Neovim? . . . iii
What This Book is not About . . . iv
What You Need to Follow Along . . . v
How to Get the Most Out of This Book . . . v
General Structure of the Book . . . vi
Notation Conventions . . . vi
The Cursor Position . . . viii
Playing Vim: The Exercises . . . viii
Appendices and Index . . . ix

Becoming a Vim Player **x**
Vim Is an Instrument . . . x
The Power Is in the Fingers . . . x
Efficient Typing: the Two Rules . . . xi
The First Week . . . xii
The Second Week . . . xii
Speed and Accuracy . . . xii
Keyboard Layouts . . . xii
Practice, Practice, Practice . . . xiii

Rank 01 - Rookie **1**
A Modal Editor . . . 1
The NORMAL Mode . . . 2
The INSERT Mode . . . 3
The COMMAND-LINE Mode . . . 4
What's a Command? . . . 4
Switching to COMMAND-LINE Mode . . . 4
Basic Ex Commands . . . 4
Completing Ex Commands . . . 5
Going Through the Ex Command History . . . 5
Vim's Help, Our Best Friend . . . 6
Moving Around with Motions (NORMAL Mode) . . . 7
Ditching the Arrow Keys . . . 7
Horizontal Motions . . . 8
Moving One Word at a Time . . . 8
Targeting a Specific Character . . . 9
Beginning, Middle, and End of Line . . . 10

Vertical Motions . . . 10
Undo and Redo (NORMAL Mode) . . . 12
Operators, Motions, and Text-Objects (NORMAL Mode) . . . 13
The Operators . . . 13
Operators and Motions . . . 13
Operators and Text-Objects . . . 14
Customizing Vim to Our Needs (Customization) . . . 16
The Main Configuration File: the vimrc . . . 16
First Configuration . . . 17
Clipboard Management . . . 18
The Clipboard Functionality . . . 18
Using the Operating System's Clipboard . . . 19
Improving Vim's Defaults . . . 19
Changing the Color Scheme . . . 20
The Configuration Addiction . . . 21
Debugging the Configuration . . . 22
Exercises . . . 23
Solutions . . . 27

Rank 02 - Novice **31**
More Vim Modes . . . 31
The VISUAL Mode . . . 31
The REPLACE Mode . . . 33
More Keystrokes to Switch to INSERT Mode . . . 34
Deleting In Vim (NORMAL Mode) . . . 35
Delete, Yank, and Put . . . 35
Deleting a Single Character . . . 36
Navigating Vim's Help (COMMAND-LINE Mode) . . . 37
Asking for Help . . . 37
Follow the Definition . . . 38
Finding the Correct Help Page . . . 39
Configuring (Neo)Vim: What Language to use? (Customization) . . . 41
Exercises . . . 43
Solutions . . . 46

Rank 03 - Beginner **50**
Searching in a File . . . 50
Searching for a Specific Pattern . . . 50
Searching for the Word Under the Cursor . . . 51
Searching in Action . . . 52
Count: Repeating Keystrokes (NORMAL Mode) . . . 54
Vim Messages (COMMAND-LINE Mode) . . . 56
The Message History . . . 57
Navigating the Message History . . . 57
Creating Messages . . . 58
Vim Options (Customization) . . . 59
Displaying the Value of an Option . . . 59
Setting Options . . . 60
Setting a Boolean . . . 60
Setting a Number . . . 60
Setting a String of Characters . . . 61
Setting a String with Spaces . . . 62
Resetting Options to Their Default Value . . . 62

Setting Options Interactively . . . 62
Persisting the Values of Options . . . 63
Finding Where an Option Has Been Set . . . 64
Searching Options in Vim's Help . . . 64
Some Useful Options . . . 65
Exercises . . . 67
Solutions . . . 70

Rank 04 - Aspirant **75**
Buffers, Windows, and Tab Pages . . . 75
Vim Buffers . . . 75
A Buffer Is Not a File . . . 76
The File Linked to the Current Buffer . . . 77
Creating and Deleting Buffers . . . 78
Switching Buffers . . . 79
The Buffer List . . . 79
Buffer Indicators . . . 80
Writing Buffers . . . 82
The Alternate Buffer . . . 83
The Swap File . . . 83
Vim Windows . . . 84
Opening Windows . . . 84
Moving Through Windows . . . 85
Resizing Windows . . . 85
Moving Windows Around . . . 86
Closing Windows . . . 86
Vim Tab Pages . . . 87
Creating and Closing Tab Pages . . . 88
Listing Tab Pages . . . 88
Moving Through the Tab Pages . . . 89
Moving Tab Pages . . . 89
Scrolling the Current Buffer (NORMAL Mode) . . . 90
Ranges for Ex Commands (COMMAND-LINE Mode) . . . 91
The Line Specifiers . . . 91
Ranges and VISUAL Mode Selection . . . 93
Mappings: Creating Our Own Keystrokes (Customization) . . . 94
Creating Custom Mappings . . . 94
Nested and Recursive Mappings . . . 96
Deleting Custom Mappings . . . 97
Mappings and Key Notations . . . 98
The Leader Key . . . 99
Ambiguous Mappings . . . 100
Listing Custom Mappings . . . 101
Finding Where a Mapping Has Been Defined . . . 102
Some Useful Mappings . . . 102
Exercises . . . 104
Solutions . . . 108

Rank 05 - Intermediate **117**
Vim Registers . . . 117
Specifying a Register . . . 117
Writing to a Register . . . 118
Reading from a Register . . . 118

Listing Registers . . . 119
The Types of Registers . . . 120
Registers Automatically Written . . . 120
Read-Only Registers . . . 121
Registers Manually Written . . . 121
The Clipboard Registers . . . 122
The Black Hole Register . . . 123
The Expression Register . . . 124
More Vertical Motions (NORMAL Mode) . . . 126
Jumping to Matching Delimiters . . . 126
Jumping Paragraphs . . . 127
Moving from Method to Method . . . 128
The Jump List and the Change List (NORMAL Mode) . . . 129
The Jump List . . . 129
The Change List . . . 130
The Command-line History (COMMAND-LINE Mode) . . . 132
The Ex Command and the Search Histories . . . 132
The Command-Line Window . . . 133
Vim Plugins (Customization) . . . 134
A Plugin to Manage Plugin . . . 135
Installing New Plugins . . . 135
Vim's Help for Plugins . . . 136
Exercises . . . 137
Solutions . . . 140

Interlude 01 - Useful Vim Plugins **146**
Neovim Providers for Remote Plugins . . . 146
Plugins Extending Vim's Functionalities . . . 147
File Explorers . . . 148
Completion and Auto-Completion . . . 148
Snippets . . . 150
The Status Line . . . 150
Vim Color Scheme . . . 151

Rank 06 - Competent **152**
Repeating Changes . . . 152
The Single Repeat . . . 152
Repeating the Last Action . . . 152
Single Repeat and Delete Registers . . . 154
Repeating the Last Ex Command . . . 155
The Complex Repeat . . . 155
Recording and Applying Macros . . . 155
Appending to a Recording . . . 157
Editing a Macro . . . 158
Inserting Control Characters . . . 160
Clearing a Register . . . 161
Editing in INSERT Mode . . . 162
Moving the Cursor . . . 162
Deletions and Past Insertions . . . 162
Copying Characters from Adjacent Lines . . . 163
Indenting . . . 163
One NORMAL mode Action in INSERT Mode . . . 163
Editing in COMMAND-LINE mode . . . 164

Moving and Deleting . . . 165
Copying Text from a Buffer . . . 165
Multiple Ex Commands on One Line . . . 166
Special Arguments for Mappings (Customization) . . . 168
Most Useful Arguments . . . 168
Ex commands and Mappings . . . 169
Silent Mappings and Ex Commands . . . 170
Vimscript Expressions in Mappings . . . 171
Exercises . . . 173
Solutions . . . 177

Rank 07 - Proficient **184**
Navigating a Project . . . 184
The Current Working Directory . . . 184
Changing the Global Working Directory . . . 186
Changing the Local Working Directories . . . 187
Searching and Finding Files . . . 187
Finding Files . . . 187
Taking the Right Path . . . 188
Excluding Files from the Search . . . 189
Searching a File Under the Cursor . . . 190
Possible Performance Problems . . . 191
Formatting Text (NORMAL Mode) . . . 192
Internal Formatting Functionalities . . . 192
Display Lines and Physical Lines . . . 192
Moving Through Display Lines . . . 195
Automatically Breaking Physical Lines . . . 195
Joining Physical Lines . . . 196
Operators to Format Text . . . 196
Formatting with an External Program . . . 197
Substitution: Search and Replace (COMMAND-LINE Mode) . . . 200
General Syntax . . . 200
The Substitute Flags . . . 201
Substitution in Action . . . 201
Replacing with Newline Characters . . . 203
Using Different Separators . . . 204
Mapping and Option Scopes (Customization) . . . 205
Option Scopes . . . 205
The Possible Scopes . . . 205
Setting the Scope of an Option . . . 206
Local Options and File Types . . . 208
Local Mappings for Buffers . . . 209
Exercises . . . 212
Solutions . . . 217

Rank 08 - Seasoned **226**
Searching in Multiple Files . . . 226
The Internal Search Engine . . . 226
Vimgrep . . . 226
Flags for Vimgrep Patterns . . . 228
Grep the Help . . . 229
Using an External Program . . . 229
Indenting Text (NORMAL Mode) . . . 233

Internal Indenting Functionalities . . . 233
Operators to Shift Text . . . 233
Customizing the Indentations . . . 233
Operator to Indent Text . . . 235
Indenting with an External Program . . . 236
Displaying the Indentation . . . 236
Charwise, Linewise, or Blockwise? (VISUAL Mode) . . . 238
The Types of VISUAL Mode . . . 238
The VISUAL Mode Charwise and Linewise . . . 239
The VISUAL Mode Blockwise . . . 240
Advanced Search in a Buffer (COMMAND-LINE Mode) . . . 242
Search and Case Sensitivity . . . 242
Displaying the Number of Matches . . . 243
Selecting and Editing the Matches . . . 244
Repeating the Last Search . . . 245
Moving from Match to Match in COMMAND-LINE Mode . . . 245
Creating Custom Functions (Customization) . . . 247
Custom Vimscript Functions . . . 247
Creating Custom Vimscript Functions . . . 247
Calling a Vimscript Function . . . 248
Listing Custom Vimscript Functions . . . 249
Finding Where a Function Has Been Defined . . . 250
Function Arguments . . . 250
Variable and Function Scopes . . . 250
Arbitrary Number of Arguments . . . 252
Line Continuation . . . 254
Custom Lua Functions . . . 255
Comparing Vimscript and Lua Functions . . . 255
Calling and Sourcing Lua Functions . . . 256
Function Scopes in Lua . . . 258
Calling Local Lua Functions in Vimscript . . . 258
Exercises . . . 260
Solutions . . . 264

Rank 09 - Believer **273**
Quickfix Lists and Location Lists . . . 273
Quickfix Lists . . . 273
The Quickfix Window . . . 274
Location Lists . . . 274
Creating Quickfix Lists . . . 275
Creating Location Lists . . . 276
Displaying and Navigating the Current Quickfix List . . . 276
Creating Quickfix Lists Using Vimscript Expressions . . . 277
Valid Entries for Quickfix Lists . . . 278
Executing Ex Commands on Quickfix List Entries . . . 279
A Stack of Quickfix Lists . . . 281
Filtering the Current Quickfix List . . . 282
Changing Case (NORMAL Mode) . . . 284
Running Shell Commands in Vim (COMMAND-LINE Mode) . . . 286
Executing Shell Commands . . . 286
Inserting Shell Command Outputs . . . 288
Filtering with Shell Commands . . . 288
Using Vim Bars with Shell Commands . . . 290

Executing Strings as Ex Commands (Customization) . . . 291
An Ex Command to Execute Strings . . . 291
Escaping Characters in Strings . . . 292
Exercises . . . 296
Solutions . . . 300

Interlude 02 - More Useful Vim Plugins **308**
A Fuzzy Finder to Find Them All . . . 308
Git Integration . . . 308
Linter Integration . . . 309
Managing Buffers and Windows . . . 309
Closing Buffers Without Closing Windows . . . 309
A New Mode to Manage Windows . . . 309
Extending Vim Motions . . . 310
Displaying an Outline . . . 310
Debugger Integration . . . 310

Rank 10 - Adept **311**
Vim Regular Expressions . . . 311
The Basics of Regular Expressions . . . 312
Vim Magical Regexes . . . 314
The Levels of Magic . . . 314
Using the Good Magic . . . 315
Vim Atom . . . 316
Word Boundaries . . . 316
Greedy and Non-greedy Quantifiers . . . 317
Vim Lookaround Assertions . . . 319
Setting the Start and End of the Match . . . 319
Lookahead and Lookbehind . . . 320
Equivalent Perl-style Metacharacters . . . 321
Matching a Regex on Multiple Lines . . . 321
Matching a Regex in a Visual Selection . . . 323
Increasing or Decreasing Numbers (NORMAL Mode) . . . 324
Copying and Moving Text (COMMAND-LINE Mode) . . . 326
Autocommands (Customization) . . . 327
Creating and Deleting Autocommands . . . 328
Listing Autocommands . . . 329
Finding Where an Autocommand Has Been Defined . . . 329
Events for Autocommands . . . 330
Multiple Events and Patterns . . . 331
Autocommand Groups . . . 331
Autocommand Local to Buffers . . . 334
Exercises . . . 337
Solutions . . . 344

Rank 11 - Veteran **353**
Vim Marks . . . 353
Creating Local and File Marks . . . 354
Listing and Deleting Marks . . . 354
Special Marks . . . 355
Marks in Action . . . 355
Absolute and Relative Line Numbers (NORMAL Mode) . . . 358
Displaying Line Numbers . . . 358

Relative Line Numbers in Action . . . 358
Switching Between Absolute and Relative Line Numbers . . . 360
Running Ex Commands on Specific Lines (COMMAND-LINE Mode) . . . 362
The Global Ex Commands . . . 362
The Global Ex Commands in Action . . . 363
Running Multiple Ex Commands . . . 364
Global Ex Commands and Substitutions . . . 364
User Commands (Customization) . . . 365
Creating User Commands . . . 366
Listing User Commands . . . 366
Finding Where a User Command Has Been Defined . . . 367
The Optional Attributes for User Commands . . . 367
Handling Arguments for User Commands . . . 368
User Commands and Completion . . . 370
Ranges for User Commands . . . 370
Local User Commands . . . 372
Exercises . . . 374
Solutions . . . 379

Rank 12 - Expert **389**
Undo in Depth: Navigating Through Changes . . . 389
The Undo Trees . . . 389
Switching Node . . . 390
Switching Branch . . . 391
The Undo Trees in Action . . . 391
Displaying Undo Trees . . . 394
The Undo List . . . 395
External Plugins . . . 396
Traveling Through Time . . . 396
Persisting Undo Trees . . . 397
Creating Undo Nodes in INSERT Mode . . . 398
Vim Abbreviations (INSERT Mode) . . . 400
Creating and Deleting Abbreviations . . . 400
Listing Abbreviations . . . 401
Finding Where an Abbreviation Has Been Defined . . . 402
Abbreviations in Action . . . 402
Fixing Typos Automatically . . . 403
Abbreviations for Frequently Inserted Text . . . 403
Expanding Abbreviations on Demand . . . 403
Local Abbreviations . . . 404
Expanding Abbreviations with Vimscript Expressions . . . 405
Executing NORMAL Mode Keystrokes (COMMAND-LINE Mode) . . . 406
The Normal Ex Command . . . 406
Executing NORMAL Mode Keystrokes in Action . . . 406
Special Keys with the Normal Ex Command . . . 407
Creating Custom Motions and Text-Objects (Customization) . . . 408
The OPERATOR-PENDING Mode . . . 409
Mappings for the OPERATOR-PENDING Mode . . . 409
Creating a Motion . . . 409
Creating a Text-Object . . . 410
Motions, Text-Objects, and VISUAL Mode . . . 411
Exercises . . . 414
Solutions . . . 420

Rank 13 - Champion **430**
Vim Folds . . . 430
Managing Folds Manually . . . 431
Creating, Opening, Closing, and Deleting Folds . . . 431
Nesting Folds . . . 432
Fold Levels . . . 433
Enabling and Disabling Folding . . . 434
Saving Folds . . . 435
Creating Folds Automatically . . . 435
Folding Methods . . . 435
Folding with Markers . . . 436
Folding by Indentation . . . 437
Folding Using Vimscript . . . 438
Moving Through Folds . . . 441
Inserting Special Characters (INSERT Mode) . . . 442
Listing Special Characters, Digraphs, and Unicode Numbers . . . 443
Inserting Special Characters . . . 443
Creating Digraphs . . . 444
Displaying Information About the Character Under the Cursor . . . 445
Finding the Digraph We Need . . . 445
Special Keywords for Ex Commands (COMMAND-LINE Mode) . . . 446
Keywords Expanding to the Text Under the Cursor . . . 446
Keywords Expanding to File Paths . . . 447
File Path Modifiers . . . 447
Expanding Keywords in Vimscript Expressions . . . 448
The Viminfo or ShaDa Files (customization) . . . 450
The Correct File for the Correct Editor . . . 451
Defining What We Save . . . 451
Exercises . . . 454
Solutions . . . 459

Interlude 03 - Vim's Startup and Runtime Paths **470**
Vim's Startup . . . 470
Startup Overview . . . 470
Running Ex Commands at Startup . . . 471
Bugs and Performance . . . 471
The Runtime Paths . . . 472
Listing All Runtime Paths . . . 473
How Vim Finds the Scripts It Needs . . . 473
Runtime Paths of Interest . . . 473
The Default Runtime Path . . . 474
The User Runtime Path . . . 476
The After Runtime Path . . . 476
The Subdirectories of the Runtime Paths . . . 477
Global Scripts Sourced at Startup (plugin) . . . 477
Filetype Detection (ftdetect) . . . 478
Filetype Scripts (ftplugin) . . . 479
Filetype Indentation (indent) . . . 481
Loading Functions On Demand (autoload) . . . 481
Creating Namespaces for Custom Functions . . . 482
Listing All Functions in a Namespace . . . 484
Syntax Highlighting (syntax) . . . 484
Color Schemes (colors) . . . 485

Selecting and Running External Tools (compiler) . . . 485
The Spell Files (spell) . . . 485
Packages of Plugins (pack) . . . 486
Vim's Help (doc) . . . 486

Rank 14 - Master **487**
Running External Applications in Vim . . . 487
Selecting an External Application . . . 488
Running an External Application . . . 489
Parsing the Output: the Error Format . . . 491
Basic Error Format Items . . . 492
Ignoring Part of the Output . . . 493
Quickfix Entry on Multiple Lines . . . 493
The Error Format in Action . . . 494
Creating a Compiler Script . . . 495
Text Completion in Vim (INSERT Mode) . . . 498
The Generic Completion . . . 499
The Completion Sub-Mode . . . 499
The Completion Menu . . . 501
The Omni Completion . . . 501
Quick Access to Documentation (NORMAL Mode) . . . 504
Using an External Application . . . 505
Using Ex Commands . . . 506
Opening Vim's Help in Scripts . . . 506
Using a User Command . . . 507
Debugging Vim's Config (Customization) . . . 508
Increasing Verbosity . . . 509
General Verbosity . . . 509
Verbosity for a Specific Ex-command or Function . . . 510
The DEBUG Mode . . . 511
Starting DEBUG Mode . . . 511
Managing Breakpoints . . . 512
The Prompt in DEBUG Mode . . . 513
DEBUG Mode Commands . . . 513
Debug Mode in Action . . . 514
Exercises . . . 517
Solutions . . . 522

Rank 15 - Grand Master **536**
The Diff Mode: Differences Between Buffers . . . 536
Enabling Diff Mode . . . 536
Jumping from Diff to Diff . . . 538
Resolving Diffs . . . 538
The Options Set in Diff Mode . . . 539
The Diff Folding Method . . . 540
Updating Highlighting and Folding . . . 540
Customizing Diff Mode . . . 541
The Default Values . . . 541
Changing Diff Algorithm . . . 541
Folding Frequency for Identical Blocks . . . 542
Ignoring Whitespaces and Case . . . 542
The Diff Mode in Action . . . 543
Resolving Diffs . . . 544

A Better Diff Display . . . 546
Advanced Macros (NORMAL & VISUAL Modes) . . . 549
VISUAL Mode Macros . . . 550
Saving and Loading Macros on Demand . . . 551
Recursive Macros . . . 552
The Arglist (COMMAND-LINE Mode) . . . 554
Displaying the Arglist . . . 555
Adding or Deleting Buffers . . . 555
Jumping to Buffers . . . 556
Running Ex Commands on Each Buffer . . . 557
The Arglist in Action . . . 557
The Status Line and the Tab Page Line (Customization) . . . 559
Customizing the Status Line . . . 559
Displaying the Status Line . . . 559
Defining the Line Items . . . 560
Customizing the Status Line in Action . . . 563
Customizing the Tab Page Line . . . 565
Displaying the Tab Page Line . . . 565
Defining the Line Items . . . 566
Customizing the Tab Page Line in Action . . . 566
Exercises . . . 571
Solutions . . . 577

Interlude 04 - Vim's File Explorer **596**
Getting Some Help . . . 597
Disabling the File Explorer . . . 597
Displaying the File Explorer . . . 597
In the Current Window . . . 597
In a Split Window . . . 598
In a New Tab Page . . . 599
As a Project Drawer . . . 599
Browsing Directories . . . 599
Opening Files and Directories . . . 600
Customizing the Display . . . 600
Filtering Files and Directories . . . 600
Navigating the Browsing History . . . 602
The Browsing History in Action . . . 603
Customizing the Browsing History . . . 604
Marking Files . . . 604
Marking Files and Directories . . . 604
Displaying Marked Files . . . 605
Running Commands on Marked Files . . . 606
Marked Files in Action . . . 606
Managing Files and Directories . . . 607
Creating and Deleting . . . 607
Renaming and Moving . . . 608
Copying and Moving . . . 608
Managing Permissions . . . 610
Bookmarking Files and Directories . . . 610
Creating and Deleting Bookmarks . . . 610
Displaying the List of Bookmarks . . . 610
Opening Bookmarks . . . 611
Opening Files with External Applications . . . 611

Exploring Remote Hosts . . . 612
Opening Remote Directories . . . 613
Reading and Writing Remote Files . . . 614
Creating New Files on Remote Hosts . . . 614
Downloading Remote Files . . . 614
Creating Keystrokes . . . 615
Using the User Maps Function . . . 615
Using a Filetype Plugin . . . 617

Rank 16 - Hero **618**
Vim Spell Checker . . . 618
Enabling the Spell Checker . . . 618
The Spell Files . . . 619
Highlighting Spelling Mistakes . . . 620
Fixing Spelling Mistakes . . . 620
Jumping to Misspelled Words . . . 622
Adding Words to Spell Files . . . 622
The Add Spell File . . . 623
Adding Words to Spell Files in Action . . . 624
Compiling Spell Files . . . 625
Cleaning up Spell Files . . . 626
Ex Command Redirections (COMMAND-LINE Mode) . . . 627
Redirecting an Ex Command Output . . . 627
Redirection in Action . . . 628
Improving Vim Performance (COMMAND-LINE Mode) . . . 631
Starting and Stopping Vim Profiling . . . 631
Profiling in Action . . . 632
Profiling a vimrc . . . 632
Profiling a Custom Function . . . 633
Profiling Everything . . . 634
Syntax Highlighting (Customization) . . . 635
Colors Supported in the Terminal . . . 636
Color Schemes . . . 636
Enabling and Disabling Syntax Highlighting . . . 636
The Syntax Files . . . 637
Syntax Items and Syntax Groups . . . 638
Listing Syntax Items . . . 639
Syntax Item Under the Cursor . . . 639
Listing Syntax Items in Action . . . 639
Creating Syntax Items . . . 640
Highlight Groups . . . 642
Listing Highlight Groups . . . 643
Creating Highlight Groups . . . 643
Resetting Highlight Group Attributes . . . 644
Highlight Groups in Action . . . 644
Linking Highlight Groups . . . 646
Finding Where a Highlight Group Has Been Defined . . . 647
Creating New Syntax Highlighting in Action . . . 647
Performance of Syntax Highlighting . . . 648
Performance Issues . . . 648
Profiling Syntax Files . . . 648
Exercises . . . 650
Solutions . . . 654

Rank 17 - Composer **668**
Navigating Codebases . . . 668
Vim Tags . . . 668
Generating Tags . . . 668
Jumping to Definition . . . 669
Completion Using Tags . . . 670
Generating Tags Files Automatically . . . 671
The Include-Search . . . 671
Matching Included Files . . . 671
The Include-Search in Action . . . 673
Searching for Defined Identifiers . . . 674
Using Included Files for Completion . . . 677
The Language Server Protocol (Neovim) . . . 677
Saving Settings and Vim Sessions (COMMAND-LINE Mode) . . . 678
Saving Options and Mappings in a File . . . 678
Saving Vim Sessions in Files . . . 679
Saving and Loading Vim Sessions . . . 679
Determining What to Save . . . 680
Managing Plugins Natively (Customization) . . . 681
Vim's Package Manager . . . 682
Installing New Plugins . . . 683
Loading Plugins on Demand . . . 684
Updating Plugins . . . 685
Using Git Submodules . . . 686
Exercises . . . 690
Solutions . . . 694

Appendices **707**
Vim's Help Indexes . . . 707
Vim's Help Quick References . . . 707
Executing Ex Commands on Lists . . . 709
The Verbose Ex Command . . . 709
Options Setting External Programs . . . 709
Events for Autocommands . . . 710

Acknowledgments

To my mom, who always encouraged me and supported me in everything I wanted to do in my life, including writing. What she did for me can't be described in words.

The biggest thanks to my girlfriend Antje, who always supported me in my weirdest dreams, in the bright and dark moments, without judgment.

To everybody who supports and actively develops both Vim and Neovim. You're true heroes.

To all my friends, to everybody who supports my work in any way, and to all the kind, peaceful, and merciful people in this world: thanks a lot!

To Bram Moolenaar, who created Vim and changed my life forever as a writer and developer. Rest in peace; I hope there are different modes in heaven.

And finally, thank you, dear reader, for picking up this book. I hope you'll enjoy it!

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. We just need to put what we know about text editors aside for a short while; Vim works differently indeed, and that's why it's so powerful.

It's easy to learn the basics of Vim; the first three chapters of this book is enough for you to edit any text file. But, because of the sheer amount of native functionalities, it takes time to master. This is also a feature, not a downside.

To me, Vim is the gamification of writing: I can focus on my writing 90% of the time, and I can 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 try to have fun doing so!

This book can be read by any Vim enthusiast out there, from the beginners who want to write their autobiographies in 25 tomes, 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, we need to clarify something important right out of the gate: Vim is a plain text editor, not a word processor.

Vim Is Not a Word Processor

A text editor like Vim allows us to write and edit plain text: the text displayed on our screen is mostly what we save in our files. A text editor is first and foremost about content, not presentation.

In contrast, a word processor, it handles content and presentation for us: we can write our text, but we can also make it bold, italic, or justified for example. LibreOffice or Microsoft Word are popular examples of word processors.

To manage this presentation, a word processor often uses a markup language which is not shown to the user. We see bold and italic text on our screen, but the file we save is

often full of tags. For example, here's an extract of a file I created in LibreOffice, and then opened in Vim:

```
<office:body>
    <office:text>
        <text:p text:style-name="P1">This is text is bold and
        ↪  italic.</text:p>
    </office:text>
</office:body>
```

The big advantage of a word processor: we don't have to care about the technical details of the markup language it uses; we just have to use the interface of the application to manage the presentation of our text. But it has also a big disadvantage: its markup language is often not compatible with any other tool, like other word processors. Even if LibreOffice developers put a lot of efforts to be able to edit Microsoft Word files in their application, there can be incompatibilities which are difficult to solve for the user.

On the other hand, there are many tools out there (especially CLI tools running in a terminal) which can manipulate plain text. There is no proprietary markup language involved, and so we can plug these tools to Vim to increase its power.

In practice, if we're writing something which needs some kind of presentation, we can use Vim to write our content as well as the markup language managing that presentation. We don't have to type everything ourselves; Vim has many functionalities and plugins to ease that burden. We can also thread our text through a whole pipeline of tools if we want this content to be presented in different formats, like PDF, HTML, or EPUB.

In short, with a plain text editor like Vim, we need to do everything ourselves. But it also means that we can use more specific and powerful tools; a word processor can't even remotely compete with those.

This book is a great example: it has been written in Neovim using Markdown, a lightweight markup language. Yet, its presentation is quite unique, specifically designed for the goal at end: learning to play Vim. It would have been nearly impossible to get the same result using a word processor.

What to Use: Vim or Neovim?

Vim is itself greatly inspired by other text editors which came before it, especially vi; Neovim is considered by many the next iteration in this long history.

As a result, both Vim and Neovim are very similar, while having also noticeable differences.

Throughout this book, the name "Vim" is used to refer to both Vim and Neovim. When some specific functionalities don't work quite the same in both editors, this book tries its best to point it out.

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

1. To configure Vim, we need to use the scripting language Vimscript. With Neovim, we can use Vimscript or Lua.
2. Neovim has some sane defaults, while Vim needs to be configured a bit more extensively.
3. Vim can be installed with or without specific features, while Neovim include most of these features by default, like clipboard management.
4. There are more tools for software developers available natively in Neovim: an LSP client for example.

Also, Neovim has tendency to change 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, some of it might not work anymore.

All in all, I would recommend using Neovim, only to be able to use some Lua for more advanced configuration. If you want to write prose instead of code, Vim is also an excellent and stable choice.

This book provides both Vimscript and Lua for most configuration. If you stick to Vimscript, you can easily switch from Vim to Neovim if you change your mind throughout 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 come back to all of that in more detail in the book.

What This Book is not About

This book is mostly about Vim (or Neovim) running in an UNIX-like shell, in a terminal. I strongly believe that Vim is most powerful in this context. As a result, this book doesn't cover Vim with GUIs (Graphical User Interfaces). That being said, most of this book is still relevant for anybody using a GUI, but not all.

You're also at the wrong address if you want to learn Vimscript 9 (a major revamp of Vimscript introduced in Vim 9, but not available in Neovim): this book doesn't speak about it, mainly because it's not widely used at the time of writing.

Also, this book doesn't cover the shell and its functionalities. If you're interested to learn more about it, and, more generally, about the tools you can use to create a Mouseless Development Environment, it's the best moment for me to shamelessly plug my other book. It's my life mission to spread the love of the keyboard.

Vim and Neovim can be extended with external plugins. While this book gives a couple of recommendations in that regard, its goal is first and foremost to cover Vim's most important native functionalities.

Finally, this book is not about re-creating your favorite IDE (Integrated Development Environment) in Vim. Vim coupled with the power of the shell is, in my opinion, even better than any IDE out there.

What You Need to Follow Along

As I was writing above, this book is specifically about Vim running in a terminal: to run the editor, you need a shell like GNU Bash or Zsh for example. If you're using macOS or any Linux-based system, you already have a shell. If you're using Windows, there are UNIX-like shells you can also install and use there.

We'll also use a couple of CLI tools directly in Vim. Here are the ones we use most often:

- grep
- find
- curl

You also need the book companion available on GitHub if you want to follow along. You can clone it directly in your shell using the CLI tool Git, or you can download it directly from the GitHub repository (click on the "code" button, and then on "Download ZIP").

How to Get the Most Out of This Book

Allow me to give you some recommendations for this book not to fail your expectations. 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.

First, this book provides practical examples from the book companion; when some files are mentioned, you can open it and experiment with all the commands and keystrokes mentioned. Active learning is key here: it's better to experiment in order to learn effectively, instead of reading passively without trying anything.

This book go through many native functionalities, and try to show how to customize them according to your own needs. Depending on what functionalities we cover in each chapter, we add more and more config as we move forward. As a result, the best is to begin this book with no config at all.

If you already load some config during Vim's startup, it shouldn't be a big problem if you understand most of it; but if you have a pre-configured Vim without understanding its config, you'll have a hard time to follow along, or even to learn about Vim's native functionalities in general. To be clear, it's fine to use a pre-configured Vim if it serves you well; but it doesn't help to learn how to customize Vim according to your specific needs.

You can also create your own cheat sheet, adding the keystrokes you're learning as you follow this book along. Again, it's better than downloading a cheat sheet made by somebody else: it can help you consolidate the newly acquired 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 cheat sheet 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 Infinite 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.

General Structure of the Book

This book is not divided in chapters but in "ranks", to give an impression of progression as we move along in Vim Land, going from the most basic concepts to more advanced ones.

Each rank build on the knowledge of the previous ones. That's why there is more and more references to past ranks as we move forward. This book also refers to the future ranks quite often, for you to check them at your leisure.

The first section of each rank is often about an important functionality, covering more than one mode (we'll see Vim modes in rank 01). Each subsequent section in a rank is about one mode specifically, or about customizing Vim; it's indicated in parentheses in the title of the section itself.

Notation Conventions

Vim offers a powerful embedded help system: what we call Vim's help here. It uses specific notations for different concepts, and this book uses most of the same notations:

Notation	Description
`[part]`	A placeholder for an optional `part` of a command.
`{part}`	A placeholder for a required `part` of a command.
`<{key}>`	A key notation for a special `{key}`.
`{key1}{key2}`	A keystroke notation where we need to hit `{key1}` followed by `{key2}`
`CTRL-{key}`	A keystroke notation where we need to hold `CTRL` and hit `{key}`.

Notation	Description
`ALT-{key}`	A keystroke notation where we need to hold `ALT` and hit `{key}`.
`CTRL-{key1} {key2}`	A keystroke notation where we need to hold `CTRL` and hit `{key1}`. Then, release `CTRL` and hit `{key2}`.
`{mode}_CTRL-{key}`	A keystroke notation for a specific `{mode}`.
`'option'`	A Vim `option`.

In this book, the word "keystroke" means a key or a set of keys which should be pressed (or "hit") to have the desired effects.

Here are some examples using all the notations above:

Example	Description
`w`	Keystroke where we need to hit `w`.
`W`	Keystroke where we need to hit `W` (or `SHIFT-w`, that is, `w` uppercase).
`d$`	Keystroke where we need to hit `d`, and then hit `$`.
`:set {option}`	The argument `{option}` is mandatory.
`:[range]delete`	The prefix `[range]` is optional.
`'number'`	The Vim option "number".
`CTRL-v`	Keystroke where we need to hit `v` while holding `CTRL`.
`i_CTRL-v`	Keystroke where we need to hit `v` while holding `CTRL` in INSERT mode.
`CTRL-w s`	Keystroke where we need to hit `w` while holding `CTRL`. Then, we need to hit `s` without holding `CTRL`.
`<esc>`	The "escape" key.
`<enter>`	The "enter" key.

This book doesn't often use the keystroke notation including the mode (like `i_CTRL-v`). That being said, this notation is useful to search a specific keystroke for a specific mode in Vim's help, if that mode is not NORMAL mode.

Regarding the special character notation, Vim's help often mix uppercase and lowercase; for example, `<Esc>`, `<PageUp>`, or `<BS>`. I find that mildly annoying; that's why this book always uses lowercase for these notations. For Vim, any case is valid: `<ESC>`, `<Esc>` or `<esc>` are equivalent.

Speaking about special character notation, it's also possible to use it to express keystrokes including a modifier key like `CTRL` or `ALT`:

Notation	Description
`<c-{character}>`	A keystroke notation where we need to hold `CTRL` and hit `{character}`.
`<a-{character}>`	A keystroke notation where we need to hold `ALT` and hit `{character}`.

For example:

Example	Description
`<c-d>`	Keystroke where we need to hit `d` while holding `CTRL`.
`<a-x>`	Keystroke where we need to hit `x` while holding `ALT`.

These last notations are most often found in mappings (see rank 04). We can't use them to search for a specific keystroke in Vim's help.

This book also tries to help you memorize the different keystrokes we can use in Vim, by linking them with what they're doing. For example, to link the keystroke `w` with the concept of "word", the "w" of "word" is underlined as follows: "word". It's for you to remember that, when hitting `w` in NORMAL mode, the cursor moves 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 detail down the line. Just remember: if you have difficulties to understand the notations of this book, you can always come back here to lighten your frustration.

The Cursor Position

The cursor is represented as a black square in this book, indicating where we should put the cursor in the different examples or exercises. For example:

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

Your specific cursor might not be represented as a block `▉` in Vim, but as a line `|`. In that case, simply put your cursor just before the letter highlighted as follows:

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

Playing Vim: The Exercises

Here are some general rules regarding the exercises in this book:

1. The "playtimes" at the end of each section are the easiest exercises. They're there to summarize their respective sections.

2. The exercises at the end of a rank dive a bit deeper in each section, or combine different sections of that rank.
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 might need to do some research in order to complete them.
4. Always revert the changes you made in the files of the book companion after finishing an exercise, simply because each exercise assume that all these files are in their original states.
   If you have Git installed on your system, running the following shell command in the root directory of the book companion should be enough:

   ```
   git checkout *
   ```

5. The solutions are at the very end of each rank.

This book tries to give you the most accurate solutions for each exercise depending on their ranks, but it's likely that there are other (and better) ones.

Don't feel pressured to complete the exercises by yourself: you can jump directly to the solutions to have more practical examples, as well as some more inspiration to configure Vim as you see fit.

There is a lot of knowledge in these pages. I wouldn't recommend trying to memorize everything; focus on what's useful for you. If you're unsure what's most relevant, the exercise ("playtime") at the end of each section summarize the most important points.

Appendices and Index

Finally, this book provides two different lists of references:

1. The appendices are useful lists we can access quickly while working with Vim.
2. The index at the end of the book only lists concepts specific to Vim or to programming in general, linking to the section of the book where the word is defined. Not every possible keystroke or Ex command is indexed, to avoid this book becoming the largest monolith in human history.

Enough rambling! Come with me, let's begin the greatest adventure in Vim Land together.

Becoming a Vim Player

Before diving into Vim itself, it's important to understand how it can best help us in our writing or in our coding. It's also a fitting opportunity to justify the title of this book.

Vim Is an Instrument

A pianist doesn't actively think about all the notes available when she's playing. Instead, if she's sufficiently trained, her movements are automatic 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: your writing.

The concept of a piano is simple to understand. If we hit a key, we hear a note. Yet, we can build more complex patterns from these simple keys, like chords, scales, or melodies. It's similar with Vim: it's quite easy to use, but hard to master. We can compose Vim's basics commands to create more complex and powerful ways to write and edit our texts.

Also, like an instrument, Vim is fun to use. It's always rewarding to find new ways to combine Vim fundamentals in elegant ways to increase our efficiency.

Vim, like many other musical instruments, only needs one interface: the keyboard. It allows us to focus on what we really want to write without anything else getting in the way. This is an important point, so let's lean into it.

The Power Is in the Fingers

We don't need to spend our time pointing and clicking with our mouse while using Vim. This statement might look weird, since it feels normal for many of us to use a mouse to write some text. But, after a decade of using Vim at this point, I can only say that using the keyboard and ditching the mouse is a blessing.

When you're 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.

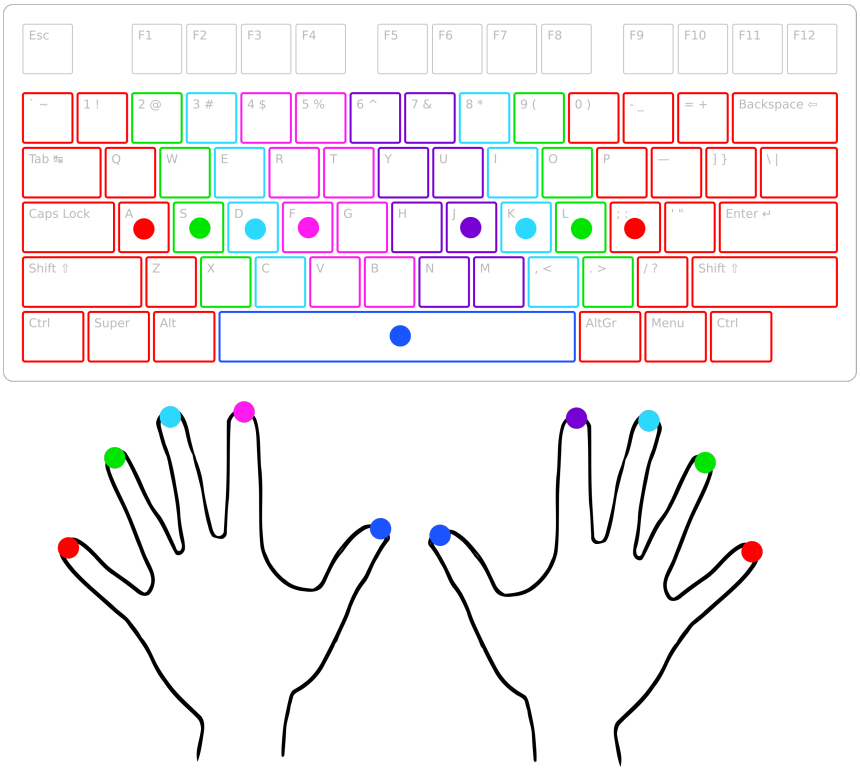
You can practice the techniques we'll see next while reading the rest of the book. It will be difficult at the beginning, but stick to it as much as you can; you won't regret it.

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 can grab any other key as efficiently as possible.

If you have a somewhat standard keyboard, you might have noticed that there are little bumps on the keys `f` and `j`. They are indicators for us to know where we need to put our indexes. When they're at the good position, simply place the other fingers on the other row keys, as shown on this image:



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. The goal is to train your muscle memory.

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 to 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 where learning happens.

Fortunately, at the end of the week, the amount of mistakes 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 anymore to look at your keyboard while typing. 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

When I was describing the row keys above, I was using the US international keyboard layout. 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, like the Escape key. Vim users hit this key again and again, so it's great to be able to reach it with our cute pinky. 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 pure joy; I can't recommend it enough.

Practice, Practice, Practice

As always, to learn as fast as possible, you need to practice. This book asks you to use your keyboard extensively, so you just need to actively follow along. We'll come back to this idea.

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 legendary country of Vim Land. It's on these beaches that we'll look at the most important and fundamental features. If you're a complete beginner and you try to actively learn the concepts explained in this first rank, you'll be able to use Vim for writing and editing your precious text right away.

More specifically, we'll cover the following in this rank:

- The most important Vim modes.
- How to move the cursor around, only using the keyboard.
- What operators, motions, and text-objects are.
- How to undo and redo changes.
- The basics of Vim customization.

To follow along, I recommend opening the file `functions.lua` from the book companion in Vim and experimenting with the different examples given in this rank. To do so, move to the root directory of the book companion in your favorite terminal, and run one of the following shell commands depending on whether you use Vim or Neovim:

```
vim functions.lua
nvim functions.lua
```

I see that you're bursting with anticipation: let's not wait further; the beaches of Vim Land only ask to be explored.

A Modal Editor

Vim is a modal editor: depending on the mode we're in, the different keys we hit on the keyboard perform different actions. This is the first big difference with most mainstream text editors, and arguably one of the main reason why Vim is so beloved and powerful.

Here are the four most important modes:

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

4. The VISUAL mode.

Let's now explore the first three; we'll look at the VISUAL mode in rank 02.

The NORMAL Mode

We're all used to be able to type the text we want to write directly after opening a random text editor. 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. The first thing you see is a welcome message. Now, if you try to hit the letter "x" on your keyboard, no "x" appears on the damn screen.

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

Let's say we want to replace a portion of some existing text using a more mainstream text editor. To do so, we would often need to:

1. Use the mouse and move our pointer to the portion of text we want to edit.
2. Click on the mouse to move our cursor (or to select some text).
3. Write whatever we want using our beloved keyboard.

This way of working is quite intuitive, but it's 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 our mouse to move our cursor, we could:

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

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 this last example. It's better for our 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: it's not meant to write any text, but instead every single key on our keyboard is a keyboard shortcut, offering us many different commands to move our cursor and target the exact piece of text we want to edit.

In other words, Vim's NORMAL mode is a keyboard-centered way to control our editor, *telling* Vim what we want to do; in return, Vim obeys our mighty wishes.

But if there are even more NORMAL mode commands than keys on a keyboard, how is it possible to memorize them all? Believe it or not, these commands make sense in Vim (most of the time), compared to the usual and somewhat meaningless shortcuts we can find in other text editors.

Vim's NORMAL mode commands use mnemonics for us to build strong muscle memory via our keyboard. Even better: these commands are composable; we can combine some of them logically. A bit like a pianist would use different notes on a scale to compose a melody.

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

The INSERT Mode

It's time to hit the first NORMAL mode command of this book, to switch 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, the indicator `-- INSERT --` is now displayed in the command-line area, at the bottom left corner of Vim.

Welcome to INSERT mode!

You can now type the text you always wanted to bring into the world, as you would do when opening most other text editors. At the end, Vim is not that different.

Now, let's try to hit the "escape" (`<esc>`) key. The indicator `-- INSERT --` disappears.

Welcome back to NORMAL mode!

This little dance is exactly what a Vim user does during a writing session: switching back and forth between NORMAL mode (to edit the existing text) and INSERT mode (to insert some text).

The NORMAL mode command `a` can also let us switch to INSERT mode, but it also moves the cursor after the current character. Try to hit `i` and `a` from NORMAL mode to see the difference. Remember: to switch back from INSERT to NORMAL mode, simply hit `<esc>`.

As you can see, Vim uses mnemonics for us to memorize these NORMAL mode commands: `i` for insert, `a` for insert after the character under the cursor.

To summarize:

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

You can see here that I use the notation `<esc>` representing the "escape" key. As we saw in the preface, 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

It's time to introduce the third most important Vim mode: the glorious COMMAND-LINE mode.

You can think of this mode as the alternative to the different menus you can find in a more mainstream text editor, where you would go to open or save a file for example.

In Vim, instead of using a mouse through layers of menus, we can instead run Ex commands for the same result (and more), without leaving our comfy keyboard.

What's a Command?

Now, you might have noticed that we are speaking about NORMAL mode *commands*, Ex *commands*, and *COMMAND*-LINE mode. So many commands!

In NORMAL mode, we can hit NORMAL mode commands; in COMMAND-LINE mode, we can execute Ex commands. The word "command" is used differently depending on the mode, and, as a result, this word becomes quite confusing.

That's why this book refers to NORMAL mode commands as NORMAL mode keystrokes. That being said, be aware that many other resources about Vim (including Vim's help) often speak about NORMAL mode commands.

Switching to COMMAND-LINE Mode

We can use the following NORMAL mode keystroke to switch to COMMAND-LINE mode:

Keystroke	Description
`:`	Switch from NORMAL mode to COMMAND-LINE mode.

When switching to COMMAND-LINE mode, the cursor move automatically to the command-line area at the very bottom of Vim, just after a colon `:`, indicating that we can write and run an Ex command.

To come back to NORMAL mode, you can hit the `<esc>` key, or simply run an Ex command.

Basic Ex Commands

Here are some basic Ex commands we all need when starting our journey through Vim Land:

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

To quit Vim, you need to quit all windows. This is not a problem here since we only have one open window; we'll see how to open more in rank 04.

You can see that most Ex commands have long and short names. The long version is more explicit, but the short version is faster to type. Writing `:write` each time we want to save a file gets old quickly; hitting `:w` is faster.

Completing Ex Commands

Using the completion in COMMAND-LINE mode can help us discover what Ex commands we can run in Vim, and what arguments we can give them.

Here are the most important COMMAND-LINE keystrokes to complete most Ex commands:

Keystroke	Description
`[pattern]<tab>`	Complete the `[pattern]` before the cursor.
`[pattern]CTRL-d`	Display all possible completions of the `[pattern]` before the cursor.

For example, if we type `:wr` in COMMAND-LINE mode and hit the `<tab>` key, Vim completes the Ex command `:write` for us. We can also complete Ex command arguments; for example, we can type `:edit <tab>` to complete file paths.

The keystroke `CTRL-d` is also essential to display all possible completions at once.

The completion in COMMAND-LINE mode is a wonderful feature; I can't recommend it enough!

Going Through the Ex Command History

Similarly to a shell (like Bash or Zsh), we can go through our Ex command history (we cover this functionality in rank 05) by using the following keystrokes:

Keystroke	Description
`<up>`	Go up in the command-line history.
`<down>`	Go down in the command-line history.

This allows us to easily go through all the Ex commands we already executed.

Vim's Help, Our Best Friend

Before switching to another topic, we need to look at the following Ex command, maybe the most important of all:

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

For example, if we want to know more about the NORMAL mode command `i`, we can run the following Ex command:

```
:help i
```

This opens a new window displaying the mythical Vim's help, showing us (hopefully) the information we seek.

I can't emphasize it enough: Vim's help is the go-to when we need any kind of information about anything Vim. If we don't remember how to quit Vim for example (like trillions of people before us), we can run the following Ex command and keep our sanity:

```
:help quit
```

That's why this book often references Vim's help at the end of sections and subsections, for us to dig deeper if we want to.

For example:



Help Yourself

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

Don't worry if the content of these pages seems too complicated at first, or if there is too much information at once. We should simply focus on using Vim as much as we can at the beginning; everything comes together with time and practice.



It's Playtime!

Exercise A – A Modal Editor

Open Vim in your terminal. Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

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's 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 memorize 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 keystrokes we can use in NORMAL mode are more easily accessible if our hands stay on the home row.

Your hands shouldn't move too much; only your fingers should. If you have a standard keyboard layout, the arrow keys are far away from the row keys, forcing you to move the 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 in NORMAL mode.

I strongly encourage you to do the same: it's a healthy and efficient way to use The Best Editor Ever Created. Also, many other CLI tools got inspired by Vim and use the same keys to move around; it's a transferable skill.

Why `hjkl`, and not some other keys close to the home row? For historical reasons. Vim is the successor of the text editor Vi, which was used on physical terminals. When you

look at the keyboard of some of them (like the Lear Siegler ADM 3A terminal), you can see that the arrow keys are the `hjkl` keys.

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

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

1. The `h` key is on the left of the sequence `hjkl`, and `l` is on the right. As a result, hitting `h` moves 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 memorize:
   - On a standard 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 beautiful art of typography, the letter "k" has an ascender, meaning that part of the letter ascend from its baseline. So `k` "ascends" our cursor.

If you're new to these ideas, I've got you covered: here's a revolutionary snake game where you have to use the `hjkl` keys to move around. If you prefer puzzle games, try this wonderful sokoban.

As always, practice will get you there.

All of that being said, the arrow keys (and other keys difficult to reach from the home row) can still have some marginal utility in Vim, especially in INSERT and COMMAND-LINE modes. The best is to have a keyboard supporting layers, to be able to switch between different layouts at the press of a key. I personally use an Ergodox, but many other keyboards offer the same functionality. Here's my layout if you're curious.

Horizontal Motions

The keys `h` and `l` are not the only ones we can use to move horizontally on the current line. In fact, efficient Vim users rarely use them. Instead, we can use other motions in NORMAL mode to move faster.

Moving One Word at a Time

It's possible to move one word at a time thanks to the following NORMAL mode keystrokes:

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

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.
- A "word" is more of a keyword composed of a specific set of characters; it mainly excludes special characters.

For example, let's open the file `functions.lua` from the book companion, and move the cursor to line 3 at the following position:

```
local function restorePosition()
```

Now, we can hit `w` and `W` multiple times to see the difference. The "WORD" motion moves the cursor beyond the parentheses `()`, but the "word" motion doesn't.

Targeting a Specific Character

Instead of moving one word at a time, we can also move to a specific character:

Keystroke	Description
`f{character}`	To find and move to the next `{character}` on the line.
`F{character}`	To find and move to the previous `{character}` on the line.
`t{character}`	To move till (just before) the next `{character}` on the line.
`T{character}`	To move till (just before) the previous `{character}` on the line.

After using one of the four keystrokes above in NORMAL mode, you can continue to move to the same `{character}` on the line with the following keystrokes:

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



Help Yourself

`:help cursor-motions`
`:help left-right-motions`
`:help Q_lr`

Beginning, Middle, and End of Line

If we want to go to the beginning, to the middle, or to the end of the current line, moving word by word can get boring quite quickly. Here are some more NORMAL mode keystrokes to accomplish these motions more easily:

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

As an aside, a whitespace is any character taking up space, but without any symbol. Practically, we mostly care about `<space>` and `<tab>` when speaking about whitespaces in Vim.

In contrast, a non-blank character, a non-whitespace character, or a non-space character is any character which is not a whitespace. All these terms are used interchangeably in Vim's help.

Vertical Motions

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

Motion	Description
`CTRL-d`	Move downward half a screen.
`CTRL-u`	Move upward half a screen.
`CTRL-f`	Move forward (downward) an entire screen.
`CTRL-b`	Move backward (upward) an entire screen.
`gg`	Move to the beginning of the current file.
`G`	Move to the end of the current file.
`{line_number}G`	Move at the beginning of the `{line_number}`.

For example, the keystroke `10G` moves the cursor to line 10.

We can also use the COMMAND-LINE mode to move to a specific line number:

Ex command	Description
`:{line-number}`	Move to the specific `{line-number}`.

For example, the Ex command `:10` moves the cursor to line 10.

Finally, here are three more keystrokes allowing us to move to the top, to the middle, or to the bottom of the current window:

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

We'll look at even more vertical motions in rank 05.



Help Yourself

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



It's Playtime!

Exercise B – Moving Around with Motions

Open the file `functions.lua` from the book companion, and 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
```

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

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 very end of the file.
5. Move to the middle of the screen.
6. Move to the first line of the file.

Undo and Redo (NORMAL Mode)

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

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

You can think of `CTRL-r` as being in control of the text.

You might 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 have inserted can be erased with one hit on `u` after coming back to NORMAL mode.

In Vim, the undo mechanism is much more powerful than in most text editors. We'll look at it in more detail in rank 12.



Help Yourself

`:help undo-redo`
`:help Q_ur`

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

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

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

The Operators

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

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

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

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

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

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

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

Operators and Motions

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

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

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

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

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

Again, if you're a beginner and you find all of these ideas complicated and confusing, I can't repeat it enough: practice like there is no tomorrow.



Help Yourself

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

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 using a motion with an operator, we operate on the text from the cursor position to 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 the 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 NORMAL mode keystrokes combining operators with text-objects:

Example	Description
`diw`	To delete inside the word. Deletes the word under the cursor.
`daw`	To delete around the word. Deletes the word under the cursor, as well as the whitespace following it.
`ciw`	To change inside the word. Deletes the word under the cursor, and switch to INSERT mode.
`yip`	To yank inside the paragraph. Yank the paragraph under the cursor.

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

In Vim's help, `daw` is described as delete a word. I find this explanation 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 this book.

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

There are many more text-objects available in Vim for our editing needs. Don't hesitate to dive deeper using Vim's help.



Help Yourself

```
:help text-objects
:help objects
:help Q_to
:help Q_tm
```



It's Playtime!

Exercise C – Operators and Text-Objects

Open the file `functions.lua` from the book companion, and 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
```

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

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`?

Customizing Vim to Our Needs (Customization)

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

You can find the final configuration we write in this section in the book companion, in the subdirectories `.vim` (for Vim) or `nvim` (for Neovim). For this last editor, you'll find the implementation in both Vimscript and Lua.

In general, most ranks in this book include a section about customization; each new addition can be found in the subdirectory of the corresponding rank.

The Main Configuration File: the vimrc

Your main configuration file can be in the following locations, depending on the editor you're using:

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

When Vim starts, it reads the configuration file `vimrc` (or `init.vim`, or `init.lua`), and executes each line. To make things simpler, the umbrella word "vimrc" is used throughout this book to refer to Vim's main configuration file, regardless of its name on our hard disk.

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. If you're a beginner, I would encourage you to write them using Vim; 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
```

Using Neovim:

```
nvim $XDG_CONFIG_HOME/nvim/init.vim
```

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 never tries 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, forcing us 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 hyphen `-` 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 have to install the package `gvim`, which also install Vim for the shell, compiled with the clipboard feature.



> **Help Yourself**
>
> `:help +feature-list` (only in Vim)

Using the 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'})
```

This line of configuration makes the copy-paste mechanism less confusing. To understand how that works, see rank 04.



> **Help Yourself**
>
> `:help clipboard`

Improving Vim's Defaults

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

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

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

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

That being said, if we only use the above Ex commands directly in Vim without writing them in our vimrc, these new mappings would disappear when we close Vim. That's

why we write them in a vimrc; because we want these Ex commands to be executed each time we start 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

" Always display the status line
set laststatus=2

" A buffer is hidden (kept in memory) if not displayed
set hidden

" Syntax highlighting
syntax on

" Enable filetype, filetype plugins, and filetype indentation
filetype plugin indent on

" Enable the package matchit
packadd! matchit
```

We basically set options to make Vim a bit more user-friendly. We'll look more closely at Vim's options in rank 03, as well as the option `'hidden'` in rank 04.

Also, you can add the following to your vimrc for Vim and Neovim if you want to display the line numbers:

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

In Lua:

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

Changing the Color Scheme

Speaking of syntax highlighting, Vim comes with many color schemes. We can switch from one to the other with the following Ex command:

Ex Command	Short name	Description
`:colorscheme {scheme}`	`:colo {scheme}`	Load the color `{scheme}`.

Remember: you can use the `<tab>` key to complete your Ex commands in COMMAND-LINE mode. So, if you want to choose among all possible color schemes available, you can simply run the following:

```
:color <tab>
```

You can also use the keystroke `CTRL-d` to display all possible color schemes at once:

```
:color <c-d>
```

Let's say we all fell in love with the color scheme "darkblue". If we want to load it each time we run Vim, we can add our Ex command above directly in our vimrc as follows:

```
color darkblue
```

Welcome to the dark ocean of Vim Land!

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.

My recommendation: 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 already available in Vim.

There is a massive time sink black hole in Vim Land: 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. Hell is not the others, it's Vim, 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, especially if you are at the beginning of your journey in Vim Land.

Debugging the Configuration

As we just discussed, 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 we've added, to understand where the problem is coming from.

Launching Vim with the following options can help in that regard:

Shell command	Description
`vim -u NONE`	Launch Vim without loading any vimrc.
`vim --noplugin`	Launch Vim without loading any plugin.

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 installed recently if the problem comes from there.

We'll look more into plugins in rank 05.

Exercises

To complete 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`, or `:q!` if you want to quit without saving!

Exercise 01 - Horizontal Motions

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
```

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

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 02 - Operators, Motions, and Text-Objects

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
```

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

1. Using only two keys, move to the first opening parenthesis as follows:

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

2. Using only three keys, delete the entire function call as follows:

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

3. Using only one key, undo your previous deletion:

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

4. Using only two keys, move your cursor to the `i` of the word `vim` as follows:

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

5. Using only one key, move to the `i` of `line` as follows:

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

Exercise 03 - Yank and Put

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
```

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

1. Hit a single key in NORMAL mode to move to the end of the line.
2. Yank the whole line.
3. Put the line yanked in step 2 above the current one.
4. Move your cursor to the beginning of the file.

Beyond the Rank

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

Exercise 04 - Yank, Delete, and Put

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
```

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

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 deleted in step 3 above the current one.
5. Put the line yanked in step 1 below the current one.
6. Using an Ex command instead of a NORMAL mode keystroke, put the line deleted in step 3 above the current one.

Exercise 05 - More Text-Objects

Move your cursor to 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
```

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

1. Using only three keys, delete the whole function `restorePosition` (spreading on 5 lines), as well as the empty line after it.
2. Using only three keys, delete the next block of parenthesis.
3. Using only three keys, delete a sentence.
4. What Ex command would you use to find out the start and the end of the text-object representing a sentence?
5. What text-object could be useful to edit some HTML? Do you think this text-object exists natively in Vim?

Exercise 06 - More Operations

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
```

Go through each step below one by one, and undo all your changes after each step to come back to the initial cursor position above.

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 functionrestorePosition()
   ```

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 to complete step 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 07 - Up and Down Following Indentations

Move the cursor to the first letter `l` 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
```

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

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

Solutions

Solution A – A Modal Editor

Step	Keystroke	Result
start		`▉`
1	`iHello Vim Land<esc>`	`Hello Vim Lan▉d`
2	`a!<esc>`	`Hello Vim Land▉!`

3. To search "vim-modes" in Vim's help, run the following 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. To quit Vim, we first need to quit the window where Vim's help is displayed with one of the following Ex commands:

   ```
   :quit
   :q
   ```

   Next, we need to quit the last window without saving our text with one of the following Ex commands:

   ```
   :quit!
   :q!
   ```

   We can also quite all windows without saving anything with one of the following Ex commands:

   ```
   :quitall!
   :qa!
   ```

Solution B – Moving Around with Motions

Step	Keystroke	Result
start		`local function ▉restorePosition()`
1	`b`	`local ▉function restorePosition()`
2	`w`	`local function ▉restorePosition()`
3	`$`	`local function restorePosition(▉)`
4	`G`	`▉}`
5	`M`	`▉]]--`
6	`gg`	`▉#!/usr/bin/env lua`

Solution C - Operators and Text-Objects

Step	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`	`local function restorePosition()`
6	`fadw`	`locfunction restorePosition()`

6. The difference between `dw` and `diw`:
   - `w` is a motion; the operation `dw` begins at the cursor position.
   - `iw` is a text-objects; the operation `diw` begins at the start of the text-object.

   That's why `dw`, in this example, delete from the cursor position to the beginning of the next word `function`, and `diw` delete inside the entire word.
   Also, a text-object always need to be prefixed by an operator. A motion can be used on its own to move the cursor.

Solution 01 - Horizontal Motions

Step	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`

Solution 02 - Operators, Motions, and Text-Objects

Step	Keystroke	Result
start		`if vim.fn.line("'\"") > 1`
1	`f(` or `t"`	`if vim.fn.line("'\"") > 1`
2	`daW`	`if > 1`
3	`u`	`if vim.fn.line("'\"") > 1`
4	`fi`	`if vim.fn.line("'\"") > 1`
5	`;`	`if vim.fn.line("'\"") > 1`

Solution 03 - Yank and Put

Step	Keystroke	Result
start		`local function restorePosition()`
1	`$`	`local function restorePosition()`
2	`yy`	`local function restorePosition()`
3	`P`	`local function restorePosition()`
4	`gg`	`#!/usr/bin/env lua`

Solution 04 - Yank, Delete, and Put

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

Solution 05 - More Text-Objects

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

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

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

4. To find out the start and end of the text-object representing a sentence, we could simply look at Vim's help by running `:help sentence` (or `:help text-object`) in COMMAND-LINE mode.
5. The text-objects `at` and `it` stand for around an HTML tag and inside an HTML tag respectively. Add an operator as a prefix to delete, change, or yank some HTML more easily.

Solution 06 - More Operations

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

3. We're using motions here: `l` is the motion to move the cursor one character to the right, `h` to move it one character to the left. Also, the motion `yl` is occasionally useful if we 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.

Solution 07 - Up and Down Following Indentations

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

Rank 02 – Novice

Welcome back, Vim explorer. We're still walking around the beaches of Vim Land in this rank, digging deeper into the functionalities we saw in the first rank.

More specifically, we'll cover the following in this rank:

- What the VISUAL and REPLACE modes are, and how to use them.
- More useful keystrokes to switch from NORMAL mode to INSERT mode.
- More NORMAL mode keystrokes to delete some text, and the consequences.
- How to efficiently use Vim's help.
- What to use to configure Neovim: Vimscript or Lua.

Take your backpack full of modes, motions, operators, and text-objects: it's time to continue our adventure in Vim Land.

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 into a bit more in this section, and, in general, in the whole book.

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

The VISUAL Mode

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

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

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

Similarly to INSERT mode, an indicator `-- VISUAL --` appears on the left bottom corner

of Vim when switching to VISUAL mode. As always, you can then use the `<esc>` key to come back to NORMAL mode.

When hitting `v` in NORMAL mode, you enter the VISUAL mode per character ("char-wise"). The selection starts at the cursor position; you can then hit some motions (or text-object) to select your text.

When some text is selected, we can then hit an operator to operate on the selection. Here are some examples:

Example	Description
`vaw`	Select visually around a word.
`vawd`	Select visually around a word, delete it, and come back to NORMAL mode. Equivalent to the NORMAL mode keystroke `daw`.
`vw`	Select visually from the cursor position to the end of the current word.
`v$`	Select visually from the cursor position to the end of line.
`vf,y`	Select visually from the cursor position to the first comma, yank the selection, and come back to NORMAL mode.
`vawyp`	Select visually around a word, yank it, and put 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 (uppercase of `v`, or `SHIFT-v`), and then hit some vertical motions to add lines to your selection.

Here are more examples:

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

There is a third VISUAL mode, where we can select blocks of text ("blockwise"); we'll see that 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 (we clearly see what's selected) make it easy to know what our operators will operate on.

That being 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 fewer keys.

Editing your text in Vim while using the fewer keystrokes possible is a challenge many Vim users like to tackle. There is 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 memorize easily, answering your specific needs, instead of obsessing on using the fewer 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 our text.



Help Yourself

`:help visual-mode`
`:help Q_vi`

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 allowing us to replace some text. Here are the keystrokes you can hit in NORMAL mode to use it:

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

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

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

The purpose of the INSERT mode is to insert new text; the purpose of the REPLACE mode is to replace existing text. It's especially useful when you don't want to mess up with some formatting.

Consider the following:

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

This is a table in Markdown, also available in the book companion if you want to play along. Let's say we want to replace the word `Stuff` under the cursor by `Keystroke`. To do so, we could hit `caw` in NORMAL mode, and then insert `Keystroke`. But the word `Description` is now misaligned:

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

Instead, if we hit `R` and type `Keystroke`, the word `Description` doesn't move, because we overwrite `Stuff` as well as the 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 us 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 useful ones:

Keystroke	Description
`i`	To insert before the current character.
`I`	To insert before the first non-whitespace character of the current line.
`a`	To insert after the current character.
`A`	To insert after the end of the current line.
`o`	To open a new line below the current one and insert some text.
`O`	To open a new line above the current one and insert some text.
`<esc>`, `CTRL-c`, `CTRL-[`	Switch back to NORMAL mode.



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, and 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
```

Undo all your changes after each step below, 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. Using only one key, open a new line below and switch to INSERT mode.
5. Using only two keys, insert a semicolon ":" 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 `d` (delete) and `c` (change). Combined with themselves, a motion, or a text-object, they're the most useful tools to erase text from existence.

But deleting in Vim can have some surprising effects when you also want to yank and paste some text. Let's explore the consequences of our deletions in this section.

Delete, Yank, and Put

Let's look at an example to illustrate one of the burdens Vim novices might stumble upon. Let's say we've the following text from the book companion:

```
local function restorePosition()
```

Let's now try the following in NORMAL mode:

1. Hit `yiw` to yank inside the word `local`. It ends up in the clipboard, to put it back later.
2. Hit the motion `w` to move to the `f` of the word `function`.

3. Hit `daw` to delete around the word `function`.
4. Hit `P` to put the text previously yanked 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 step 4. After all, many other text editors would paste the text previously copied, and not the text previously deleted.

As you can see, Vim behaves differently. To take another weird analogy, it's a bit like Vim would throw whatever we delete into the clipboard, as if it was a trash bin. It means that anything we delete is always brought back if it's the last thing we did before hitting a put keystroke (`p` or `P`) in NORMAL mode.

This last idea is not really true, but it will do for now. All of that 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 worth it.

Deleting a Single Character

There are two more NORMAL mode keystrokes we can use to delete a single character:

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

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



Help Yourself

```
:help deleting
:help Q_de
```



It's Playtime!

Exercise B – Deleting in Vim

Open the file `functions.lua` from the book companion, and move your cursor to the following position:

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

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

1. Using three keys, delete the word `restore`.
2. Using two keys, replace the uppercase `P` of `Position` with the lowercase `p`.
3. Using one key, move to the `n` of `position`.
4. Using one key, put what you've deleted, to end up with `positionrestore`.
5. Using one key, delete the last `e` of `positionrestore`.

Navigating Vim's 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's help explains almost everything, making it more complicated, more dense, but also more complete.

At the risk of sounding like a broken record, Vim's help is our most precious ally in the wonderful Vim Land: it can answer most of our questions. But to get the good answer, we first need to know how to ask properly.

Asking for Help

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

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

Also, as we saw in the preface, Vim's help use specific notations to describe keystrokes, Ex commands, and more.



Help Yourself

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

Follow the Definition

Vim's help is a set of text files. Thanks to tags files (we'll see this concept much further in the book, in rank 17), we can jump to the definitions of some specific keywords from any open help window. We can think of these tags as links, allowing us to dig deeper and deeper in this library of knowledge. We can recognize these links thanks to syntax highlighting.

If we want to follow one of these links, we can place our cursor on one of them, and hit the NORMAL mode keystroke `CTRL-]`. We would then jump to another part of Vim's help. To jump back, we can hit `CTRL-o` in NORMAL mode; we'll come back to this specific keystroke in rank 05.

Let's try it. First, let's run the following Ex command:

```
:help
```

The current window is split vertically in two. Welcome to the main help file!

Then, let's place the cursor on the word `bars` as follows:

```
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.
```

As written in this help file, hit `CTRL-]` to jump to the subject `bars`, at the end of the same file. To come back to where we were, hit `CTRL-o`.

The main help file for Neovim changed recently, but the tag `bars` should still be there.

To summarize:

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

Remember: to close any window (like a window displaying Vim's help), run the Ex command `:q` in COMMAND-LINE mode. Closing the last window closes Vim. We'll come back to this concept of window in rank 04.

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

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

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



Help Yourself

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

Finding the Correct Help Page

We've seen that we can search through Vim's help using the Ex command `:help {subject}`. This subject is also a tag. That's great and all, but how do we know what tag to use to jump where we want to jump?

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

Ex command	Description
`:help {subject}`	Search for a specific `{subject}`. A `{subject}` in Vim's help is often composed of a couple of words separated by hyphens `-`.
`:help {keystroke}`	Search for a NORMAL mode `{keystroke}`.
`:help :{ex-command}`	Search for an `{ex-command}`.
`:help CTRL-{character}`	Search for a NORMAL mode keystroke `CTRL-{character}`.
`:help i_CTRL-{character}`	Search for an INSERT mode keystroke `CTRL-{character}`.
`:help c_CTRL-{character}`	Search for a COMMAND-LINE mode keystroke `CTRL-{character}`.
`:help -{option}`	Search for the command-line `{option}` we can give to Vim when running it from a shell, in the terminal.

To drive the point home, here are some practical examples:

Example	Description
`:help insert-mode`	Search for the topic `insert-mode`.
`:help :help`	Search for the Ex command `:help`.
`:help o`	Search for the NORMAL mode keystroke `o`.
`:help daw`	Search for the NORMAL mode keystroke `daw`.
`:help CTRL-o`	Search for the NORMAL mode keystroke `CTRL-o`.
`:help i_CTRL-o`	Search for the INSERT mode keystroke `CTRL-o`.
`:help c_CTRL-f`	Search for the COMMAND-LINE mode keystroke `CTRL-f`.
`:help i_CTRL-r_CTRL-r`	Search for the INSERT mode keystroke `CTRL-r` followed by `CTRL-r`.
`:help -u`	Search for the command-line option `-u` we can give to Vim in the shell as follows: `vim -u`.

There are more patterns available for the arguments of the Ex command `:help`, to specify what we're actually searching. We'll discover them throughout the book.



Help Yourself

`:help online-help`



It's Playtime!

Exercise C - Navigating Vim's Help

Consider the following from our beloved Vim's 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's help?
3. What Ex command would you use to find the NORMAL mode keystroke `CTRL-v` in Vim's help?
4. What Ex command would you use to find the INSERT mode keystroke `CTRL-v` in Vim's help?
5. What Ex command would you use to find the COMMAND-LINE mode keystroke `CTRL-v` in Vim's 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 functions and Ex commands to customize Vim as we see fit.

But, over the years, Vimscript mutated: it became a more general scripting language. It's also where things begin to get a tad ugly; over the years, Vimscript gained some pitfalls due to weird design decisions. It can be painful to understand, use, and debug for anything too complicated.

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 goals was to provide 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 provide 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, this book doesn't cover Vim9 Script at all. Simply be aware that you can write your vimrc in Vim9 Script for Vim (but not for Neovim) if you want to.

Most of the configuration in this book is in Vimscript. It also tries to provide the equivalent scripts in Lua, but not always, especially when approaching the end of the book. At that point, I expect you, dear reader, to understand how to transcript from Vimscript to Lua if you really want to.

In my opinion, Lua is especially useful when the configuration becomes more complex. Even so, I still consider Vimscript as a good language when it comes to configuring The Best Editor in the Word.

When using Neovim, why not configuring everything in Lua at the first place? That's a great question, and so here's my subjective opinion:

1. Many good resources and useful functions available on the glorious Internet are written in Vimscript. Understanding the basics of this language can help you understand and customize what you can find online.
2. Vimscript will continue to be supported by both Vim and Neovim.
3. At the time of writing, many Vimscript functions don't have any equivalence in Lua. If you begin to write some functions in Lua for example, you'll often need to call some Vimscript functions in there, resulting in a somewhat confusing mix.
4. It's easier to use Vimscript when crafting Ex commands in COMMAND-LINE mode, and so it's useful to know some basics even without speaking about configuration.

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

Finally, 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 01 – The VISUAL Mode

Open the file `functions.lua` from the book companion, and 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 if needed.

1. Select the word under the cursor without its following `<space>`.
2. Select everything until the word `and` on the current line (without including it), and yank the text selected.
3. Without using a selection, find a shorter NORMAL mode keystroke to yank the same text as step 2.
4. Select everything inside the next parentheses on the current line.
5. Select the current line and the line below.

Exercise 02 – Help Yourself

Using Vim's help, how would you:

1. Find information about the quickfix functionality?
2. Find information about the NORMAL mode keystroke `*`?
3. Find information about the REPLACE mode?
4. Find information about the Ex command `:save`?

Beyond the Rank

These exercises are more difficult. The solutions often involve some complementary concepts not covered in the previous ranks.

Exercise 03 – The VISUAL Mode, the Return

Open the file `functions.lua` from the book companion, and 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 if needed.

1. Only using two keys, select the current line and the next two lines.
2. Select and save the current line to the file path `/tmp/file`.
3. Select the whole function, and indent your selection twice.
4. Select everything from your cursor position until the first occurrence of the word `normal` on the line below.

Exercise 04 - More Help, Please

Using Vim's help, how would you:

1. Find information about the option `'filetype'`?
2. Find information about the INSERT mode keystroke `<esc>`?
3. Find information about the Vimscript function `expand`?
4. Find information about the regex metacharacter `=`?
5. Find information about the following error message:

```
E492: Not an editor command: abcde
```

Exercise 05 - Swapping Characters, Words, and More

Open the file `functions.lua` from the book companion, and 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
```

Undo all your changes after each step below, and come back to this initial cursor position. Also, use only the NORMAL mode for each step.

1. Only using two keys, swap the character `l` with the following character `o`.
2. Only using three keys, swap the current line with the line below.
3. Only using five keys, swap the word `local` with the word `function`.

4. Only using five keys, swap the entire function `restorePosition()` with the other function `deleteTrailingWS()`.

Solutions

Solution A – Even More Vim Modes

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

Solution B – Deleting in Vim

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

Solution C – Navigating Vim's Help

1. `[range]` and `[ove]` are optional; `{address}` is mandatory.
   As a result, all following Ex commands are valid:

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

2. To find the entry above in Vim's help:

   ```
   :help :move
   ```

3. To find the NORMAL mode keystroke `CTRL-v` in Vim's help:

   ```
   :help ctrl-v
   ```

4. To find the INSERT mode keystroke `CTRL-v` in Vim's help:

   ```
   :help i_ctrl-v
   ```

5. To find the COMMAND-LINE mode keystroke `CTRL-v` in Vim's help:

```
:help c_ctrl-v
```

Solution 01 - The VISUAL Mode

1. We need to hit `viw` in NORMAL mode to select the word under the cursor without the following space.
   The keystroke `vaw` selects the word under the cursor with the following space, and `vw` selects until the first letter of the next word.
2. We need to hit `vtay` in NORMAL mode to select everything until the word "and".
   - `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. We need to hit `vi(` or `vib` to select everything inside the next block of parentheses.
   If our cursor is not already between parentheses, the text-object `i(` or `ib` always points to the next parentheses on the current line, moving the cursor to start the operation at the good location.
5. We need to hit `Vj` in NORMAL mode to select the current line and the line below.
   - `V` switches to VISUAL mode linewise.
   - `j` moves the cursor to the next line, extending the selection in the process.

Solution 02 - Help Yourself

1. To find information about the quickfix functionality:

```
:help quickfix
```

2. To find information about the NORMAL mode keystroke `*`:

```
:help *
```

3. To find information about the REPLACE mode:

```
:help replace-mode
```

4. To find information about the Ex command `:save`:

```
:help :save
```

   In general, to search for an Ex command in Vim's help, always prefix the Ex command's name with a colon `:`.

Solution 03 - More VISUAL Mode

1. To select the current line and the next two lines, we can use a count (see rank 03): hitting `3V` in NORMAL mode selects the current line as well as two more lines below.
2. We can follow these steps to select the current line and save it to a file:
   1. Hit `V` to select the current line.
   2. Hit `:` to switch to COMMAND-LINE mode.
   3. We can see the marks `'<,'>` appearing in the command-line (we'll see marks in rank 11). It means that the following Ex command acts on the selection; we can therefore run the following:

      ```
      :'<,'>write /tmp/file
      ```

   To make sure that it worked, we can simply open the file we just created as follows:

   ```
   :edit /tmp/file
   ```

3. We can select the whole function by hitting `vip` in NORMAL mode. We can then indent our selection, using the NORMAL mode keystroke `>`; here are two different ways to do it two times:
   - Use a count: hit `2>` (see rank 03).
   - Hit `>`, and then repeat the last action with `.` (see rank 06).
4. To select everything from the cursor position to the first occurrence of the word `normal` in the current file, we can hit `v/normal`; the search `/normal` can be considered a motion here.

Solution 04 - More Help, Please

1. To find information about the Vim option `'filetype'`:

   ```
   :help 'filetype'
   ```

   In general, to search for a Vim option in Vim's help, always surround the option's name with single quotes `'`.
2. To find information about the INSERT mode keystroke `<esc>`:

   ```
   :help i_<esc>
   ```

3. To find information about the Vimscript function `expand`:

   ```
   :help expand()
   ```

   In general, to search for a Vimscript function in Vim's help, always prefix the function's name with parentheses `()`.
4. To find information about the regex metacharacter `=`:

```
:help /\=
```

In general, to search for a regex metacharacter in Vim's help, always prefix it with `/\`

5. To find information about the error message E492:

```
:help E492
```

Solution 05 – Swapping

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

Rank 03 – Beginner

We're now getting further away from the beaches of Vim Land, making our way to the inner country. This is the domain of an ancient and powerful functionality, allowing us to search and match any piece of text: the glorious Vim search.

More specifically, we'll cover the following in this rank:

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

This arcane knowledge is forever useful in Vim Land. Let's put on our wizard hat without waiting, and let's begin the incantations.

Searching in a File

We can search in the current file using the COMMAND-LINE mode, or using some handy NORMAL mode keystrokes.



> **Help Yourself**
>
> `:help search-commands`
> `:help Q_pa`

Searching for a Specific Pattern

As we saw in rank 01, we can hit the keystroke `:` in NORMAL mode to switch to COMMAND-LINE mode and run Ex commands. There is another thing we can do in this exact same mode: searching.

Here are the NORMAL mode keystrokes we all need:

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

This pattern can be a regular expression, meaning that a regex metacharacter needs to be escaped to match its literal counterpart. For example, if we search for a dot `.`, we need to escape it with a backslash `\` since a dot has specific semantics in the context of regular expressions. We'll look at Vim regular expressions more in details in rank 10.

If we only want our pattern to be literal without bothering with regular expressions, we can also prefix our patterns with `\V` to make sure that it's "very nomagic", meaning that all its characters are literal. We'll explore the levels of magic for regular expressions in rank 10. I told you to take your wizard hat!

By default, when hitting `<enter>` after typing a pattern, Vim switches back to NORMAL mode, and the cursor moves directly to the first match if there is any. The cursor might even move before hitting `<enter>` depending on Vim's configuration.

If there is no match, Vim displays an error:

```
E486: Pattern not found: {pattern}
```

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

Keystroke	Description
`n`	Move to the next match.
`N`	Move to the previous match.

Searching for the Word Under the Cursor

We can also search directly for the word under the cursor thanks to the following NORMAL mode keystrokes:

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

Searching in Action

The main goal of this book is to be a practical guide to Vim. In that context, I tried to include as many examples as I could, and it's time for one of them!

Let's open the file `functions.lua` from the book companion, and let's move the cursor to line 3 at the following position:

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

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

If we hit `/` in NORMAL mode, we switch to COMMAND-LINE mode; the cursor ends up at the bottom of Vim. From there, we can type the pattern we want to search.

Let's say we want to search for the word "vim", so let's type the following:

```
/vim<enter>
```

Remember that `<enter>` refers to the key here; we shouldn't type the literal word `<enter>`. So when hitting `<enter>`, the cursor moves to the first match of the word "vim", and we switch back automatically to NORMAL mode:

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

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

We can then hit `n` in NORMAL mode to move from match to match; doing so two times moves the cursor to the last occurrence of `vim` on the current line:

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

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

That's not all: searching can be considered as a motion (see rank 01). It means that we can combine an operator with a search. Similarly to any other motion, the operator operates from the cursor position to the first match.

To illustrate this idea, let's move back to the word `function` by typing the following:

```
?fun<enter>
```

We're back on the `f` of `function`:

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

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

Let's now try to type the following:

```
d/Pos<enter>
```

It means that we delete everything from the cursor position to the first match of the pattern `Pos`. The result:

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

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

Like most functionalities in Vim, its search feature can be greatly customized to fit our specific needs. We'll see three important options to do so in this rank.

In summary, Vim search is very useful for:

1. Moving our cursor throughout the current file.
2. Being able to use an operator from the cursor position to a specific pattern.



It's Playtime!

Exercise A – Searching in a File

Open the file `functions.lua` from the book companion, and move the 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
...
```

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

1. Search forward for the word `line`, and move your cursor to the first match.
2. Using only one key, move to the next match of the word `line`.
3. Using only one key, move back to the first match of the word `line`.
4. Search backward for the word `function`, and move your cursor to the first match.
5. Using a motion, move to the letter `l` of the previous word `local`.
6. Using only one key, search forward for the pattern under the cursor.

Count: Repeating Keystrokes (NORMAL Mode)

Most NORMAL mode motions we've seen in rank 01 can be repeated; we only need to add a count (a number of repeat) as prefix.

For example, let's open in Vim our new favorite file `functions.lua` from the book companion, and move the cursor to line 3 at the following position:

```
local function restorePosition()
```

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

```
local function restorePosition()
```

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

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

```
restorePosition()
```

We've deleted two words.

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

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

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

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

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

Using counts is also more useful when the count stays low. It's easy to predict what will happen when you want to delete a couple of words, lines, or paragraphs; not so much if you try to delete 42 of them in one stroke.

A last word about count: since many NORMAL mode keystrokes can take a count as prefix, this book won't always specify when a count is possible for a specific keystroke, for readability purposes. Thankfully, Vim's help always indicate if you can use a count with the notation `[count]`.

We'll come back to this functionality throughout the book when it's most useful anyway, with, as always, tasty practical examples.



Help Yourself

`:help count`



It's Playtime!

Exercise B - Count: Repeating Keystrokes

Open the file `functions.lua` from the book companion, and 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
```

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

1. Using only three keys, move to the third letter `v` of the current line as follows:

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

2. Using only three keys, insert two new empty lines below the current one.
3. Delete in one operation the current line as well as the next two consecutive ones below.
4. Delete in one operation everything from the cursor position until the third word `fn` of the current line.

Vim Messages (COMMAND-LINE Mode)

I personally believe Vim is a sentient creature, and I've the ultimate proof: sometimes, it speaks to me.

Indeed, Vim can display some messages in the command-line area at the bottom of the screen. For example, as we saw just above, when we search for some pattern in the current file without match, Vim tells us with verve and panache. This is the undeniable proof of superintelligence right there.



Help Yourself

`:help messages`

The Message History

Good news everyone: nothing is eternal. A message displayed disappears if Vim needs to display something else in the command-line area, like another message, or even the Ex command we're frenetically typing in COMMAND-LINE mode.

That's unfortunate if we didn't have time to read the last message, or if we don't remember what Vim told us. Thankfully, it's possible to bring back *some* messages from the dead thanks to the following Ex command:

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



Help Yourself

`:help message-history`

Navigating the Message History

Now that we can display the message history, a question arise: how to navigate through these messages if the output exceed the size of the screen?

In that case, Vim uses its own internal terminal pager, similar to other pagers like less.

As an aside, speaking about pagers, one of the advantages of learning to play Vim is that many other CLI tools use the same keystrokes to some degree, including most UNIX terminal pagers. In other words, the magical knowledge contained in these very pages can be useful for many other tools!

Back to our message history, here are the most useful keystrokes we can use to navigate through our messages if the output is too big for our small screen:

Keystroke	Description
`j`	Move one line downward.
`k`	Move one line upward.
`d`	Move downward half a screen.
`u`	Move upward half a screen.
`f`	Move forward (downward) an entire screen.
`b`	Move backward (upward) an entire screen.
`G`	Move to the bottom of the message history
`gg`	Move to the top of the message history.

As you can see, these keystrokes are very similar to the ones we saw in rank 01.



Help Yourself

`:help pager`

Creating Messages

It's crucial to be able to craft our own messages, to debug a malfunctioning vimrc for example. To do so, we can use the following Ex commands:

Ex command	Short name	Description
`:echo {expression}`	`:ec {e}`	Display the output of `{expression}`, but doesn't add it to the message history.
`:echomsg {expression}`	`:echom {e}`	Display the output of `{expression}`, and add it to the message history.
`:echoerr {expression}`	`:echoe {e}`	Display the output of `{expression}` as an error message, and add it to the message history.

An expression is not an Ex command but some Vimscript. For example a string of character, a variable, or a function call.

Speaking of examples, here are some showing what we can do with the Ex commands above:

Example	Result
`:echo 'hello'`	`hello`
`:echom 'greetings' 'bonjour'`	`greetings bonjour`
`:echo 2+2`	`4`
`:echo system('ls')`	The function `system()` can run a shell command. Here, we run the CLI tool `ls` and display its output.
`:let var='hello' | echo var`	`hello`

As we can see with the last example, we can write multiple Ex commands in COMMAND-LINE mode on one line, separating them with a pipe character `|`. Here, we first define a variable `var`, and then we display its value.

We'll come back to this idea of multiple Ex commands on one line in rank 06, and we'll also quickly cover Vimscript variables in rank 08.



It's Playtime!

Exercise C – Vim Messages

Open the file `functions.lua` from the book companion, and move the cursor to the following position:

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

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

1. What message Vim gives you when trying to search for the pattern "emacs" in the current file?
2. Display the message "A fragile message" without saving it in the message history.
3. Display the message "A true message", and save it in the message history.
4. Display the entire message history, including the two messages from step 1 and 3.

Vim Options (Customization)

Vim comes with many options for us to customize The Best Editor in the Galaxy as we see fit. These options are similar in concept to the settings we would modify to customize other text editors.

We've already seen some of these options in rank 01; let's now explore how they work.



Help Yourself

`:help options`

Displaying the Value of an Option

Before modifying the value of an option, it's often useful to check the default one. You can do so with these two equivalent Ex commands:

Ex command	Description
`:set {option}?`	Display the value of the `{option}`.
`:echo &{option}`	Display the value of the `{option}`.

For example, the following Ex command displays the value of the option `'clipboard'`:

```
:set clipboard?
```

It's a bit like asking the option for its value, hence using a question mark `?`.

We can also manipulate an option like a Vimscript variable if we add the prefix `&` to its name; we'll see throughout this book that, in some situations, it can be very convenient to do so.

Setting Options

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

- A boolean.
- A number.
- A string of characters.

There are different ways to set the value of an option, partially depending on its type.



Help Yourself

`:help set-option`

Setting a Boolean

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

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

For example, we can turn on the option `'number'` to display the line numbers thanks to the following Ex command:

```
:set number
```

To turn it off:

```
:set nonumber
```

Setting a Number

We can set the value of an option accepting a number as follows:

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

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

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

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

```
:set laststatus = 0
```

Setting a String of Characters

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

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

The two last Ex commands are useful because, for some options, we can give them more than one value. The value of the option itself is still a string of characters, but separated by commas to represent the different sub-values.

For example, in rank 01, we appended a value to the option `'clipboard'` in our vimrc as follows:

```
:set clipboard+=unnamedplus
```

To delete the value we just added, we can run the following:

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

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

```
:set clipboard=unnamed
```

Setting a String with Spaces

When setting a string to an option, we have to escape whitespaces using backslashes `\`. For example:

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

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

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

For example, the two following Ex commands are equivalent:

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

We were stating above that adding the prefix `&` to an option allows us to manipulate it like a Vimscript variable. It's very useful in this specific context to avoid escaping `<space>` characters, making an option easier to set, read, and understand.

We'll see what the option `'formatprg'` stands for in rank 07.



Help Yourself

`:help :let-option`

Resetting Options to Their Default Value

It's easy to fiddle with an option and change its value, but what if we want to set the option back to its default? As always, Vim got us covered here:

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

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

```
:set laststatus&
```

The default value of an option is indicated in Vim's help.

Setting Options Interactively

There is another way to set Vim options:

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

We can then go through the entire list of options, and hit `<enter>` on the following:

- On the option's name: open Vim's help for this specific option.
- On the option's value: change the value of the option.

This Ex command is mostly useful to discover new options: there is a short help message for each option listed, and it's easy from there to access Vim's help for a specific one. That being said, setting options via this interface is often clunky, and on most of my terminals it simply doesn't really work.

The sheer number of options here can be a bit daunting; don't try to learn them all at once, or your mental health might degrade! I don't want this book to be the reason of your downfall.

Persisting the Values of Options

As we already saw, if we want anything to be persisted between Vim sessions (a Vim session begins when we open Vim, and end when we close it), we need to add it to our vimrc. Options are no exception.

If you're using Neovim and you want to set your options in the file `init.lua`, here are the equivalent Ex commands you can add to your vimrc:

Vimscript	Lua
`set number`	`vim.opt.number = true`
`set nonumber`	`vim.opt.number = false`
`set laststatus=1`	`vim.opt.laststatus = 1`
`set filetype=vim`	`vim.opt.filetype = 'vim'`
`let &formatprg='prettier %'`	`vim.opt.formatprg = 'prettier %'`
`set clipboard+=unnamedplus`	`vim.opt.clipboard:append({ 'unnamedplus' })`
`set clipboard-=unnamedplus`	`vim.opt.clipboard:remove({ 'unnamedplus' })`

Lua is not as finicky as Vimscript when it comes to spaces; while we shouldn't use spaces before or after the equal character `=` when using `:set`, Lua won't scold us because of spaces.

Remember: we don't need the prefix `:` if we want to put some Ex commands in our vimrc. Instead of writing `:set laststatus=2`, we should write `set laststatus=2`. It's a convention we all follow, a rule of the Mighty Vim.

Finding Where an Option Has Been Set

It can be useful to display where an option has been set, especially as a vimrc grows bigger and bigger. For example, let's say we've written the following somewhere in our vimrc:

```
set laststatus=2
```

We want to change this value, but we don't remember what line it's set on. We could open our vimrc and search for it, or we could also use the following Ex command to refresh our memory:

Ex command	Description
`:verbose {ex-command}`	Display more information with the output of the `{ex-command}`.

We can run the following Ex command for our specific example:

```
:verbose set laststatus?
```

Here's what I get on my machine:

```
laststatus=2
      Last set from ~/.config/nvim/init.vim line 34
```

We'll see throughout this book that `:verbose` can be used with other Ex commands to pinpoint exactly where some entities have been defined, and in general to display more information about the output of the Ex command given as argument.

Searching Options in Vim's Help

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

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

We can get this information in Vim's help, as well as an option's type, its default, and much more. To specifically search for an option, we have to surround its name with single quotes `'`. For example:

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

That's why this book surrounds a Vim option with single quotes (e.g. `'laststatus'`): to remind us this important fact.

Some Useful Options

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

Option	When turned on	Default
`'ignorecase'`	Search ignore the case sensitivity (uppercase and lowercase characters are equivalent).	off
`'smartcase'`	Search is case-sensitive if there is one or more uppercase in the pattern. Needs the option `'ignorecase'` to be on.	off
`'hlsearch'`	Highlight the matches of a search.	off (Vim) on (Neovim)
`'autowriteall'`	Automatically write open files.	off
`'incsearch'`	Highlight the matches in the current file while typing a search pattern.	on

A last detail: Vim options have also short names, similarly to Ex commands. For example, you can turn on the option `'ignorecase'` with one of the following equivalent Ex commands:

```
:set ignorecase
:set ic
```

That being said, I'd encourage you to always write an option's long name in your vimrc, for better readability. The option `'ignorecase'` is quite self-explanatory, `'ic'` much less so. This rule can also apply to Ex commands.

We're only at the beginning of our relationship with Vim options. We'll have the occasion to develop deep friendship with many of them throughout this book. We'll also cover the different scopes an option can have in rank 07.



Help Yourself

`:help option-list`



It's Playtime!

Exercise D – Vim Options

Open the file `functions.lua` from the book companion.
Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

1. Display the value of the option `'filetype'`.
2. Toggle the value of the option `'number'`.
3. Append the sub-value "S" to the current value of the option `'shortmess'`.
4. Delete the sub-value "S" of the current value of the option `'shortmess'`.
5. Set the value of the option `'shortmess'` back to its default.
6. Find out the short name of the option `'clipboard'`.

Exercises

Exercise 01 – The Power of Vim Search

Open the file `functions.lua` from the book companion, and 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`"]])
...
```

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

You should also use Vim search for each step.

1. Move your cursor to the following position:

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

2. Move your cursor to the following position:

   ```
   vim.cmd([[normal! g`"]])
   ```

3. Move your cursor to the following position:

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

4. Yank everything from the cursor position until the word `then` at the end of the line.
5. Delete everything from the cursor position until the word `then` at the end of the line.

Exercise 02 – Search Highlighting

Open the file `functions.lua` from the book companion, and move the cursor to the following position:

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

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

1. Search for the first occurrence of the word "vim", and move your cursor to its loca-

tion.

2. Turn off the highlighting of the matches of a search.
3. Turn on the highlighting happening when typing a matching search pattern.
4. Toggle both search highlighting from step 2 and 3. What search highlighting is turned on or off?

Exercise 03 – Count in Action

Open Vim without opening any file.

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

You should also use a count for each step.

1. Insert four times the word "vim" separated by `<space>` characters, and come back to the beginning of the line.
2. Select the first three words on the line, including the spaces separating them; then, cancel the selection.
3. Yank the current line and copy it three times below.
4. In one operation:
    1. Create four new lines.
    2. Insert a star character `*` at the beginning of each odd line.
    3. Let the even lines blank.

Beyond the Rank

These exercises are more difficult. The solutions often involve some complementary concepts not covered in the previous ranks.

Exercise 04 – Vim Messages

Open the file `functions.lua` from the book companion, and 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
```

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

1. Display the output of the shell command `file` with the relative path of the current file as argument, without saving it in the message history.
2. After pressing `<enter>` to hide the display of the previous step, display it again.
3. Try to search for the word “emacs” in the current file.
4. Clear all messages from the message history.
5. Display the error message you should have got in step 3.
6. Write an entry in the message history including the following, separated by pipe characters `|`:
    1. The full path of the current file.
    2. The `'filetype'` of the current file.
    3. The current line number.
    4. The word under the cursor.

Exercise 05 – Setting Vim Options

Work through the steps below one by one, preserving your changes after each step. Restore all files to their original states at the end of the exercise.

1. Verify that the option `'shortmess'` can have a local value.
2. Turn off the option `'expandtab'`, but only for the current buffer.
3. Set the option `'listchars'` to the following value, only for the current window:

    ```
    tab:> ,trail:±
    ```

    Note that there is a space between the characters `>` and `,`.
4. In Vim (not in Neovim), find the two options allowing us to have a cursor block in NORMAL mode `▮`, and a cursor line `|` in INSERT mode.

Solutions

Solution A – Searching in a File

Step	Keystroke	Result
start		`#!/usr/bin/env lua`
1	`/line<enter>`	`if vim.fn.line("'\"") > 1 and vim.fn.line`
2	`n`	`if vim.fn.line("'\"") > 1 and vim.fn.line`
3	`N`	`if vim.fn.line("'\"") > 1 and vim.fn.line`
4	`?function<enter>`	`local function restorePosition()`
5	`b`	`local function restorePosition()`
6	`*`	`local function deleteTrailingWS()`

Solution B – Count: Repeating Keystrokes

Step	Keystroke	Result
start		`if vim.fn.line("'\"") > 1`
1	`3fv`	`<= vim.fn.line("$") then`
2	`2o<esc>`	
3	`3dd`	`end`
4	`3d/fn`, `d3/fn`, `4dtf`, or `2d2tf`	`fn.line("$") then`

Solution C – Vim Messages

1. After hitting `/emacs<enter>`, Vim displays the following error message:

   ```
   E486: Pattern not found: emacs
   ```

2. To display a message without saving it in the message history:

   ```
   :echo "A fragile message"
   ```

3. To display a message and save it in the message history:

   ```
   :echom "A true message"
   ```

4. To display the message history:

   ```
   :mes
   ```

Solution D – Vim Options

1. To display the value of the option `'filetype'`:

   ```
   :set filetype?
   ```

   The value of `'filetype'` should be `lua`, since we have a Lua file open.
2. To toggle the value of the option `'number'`:

   ```
   :set number!
   ```

3. To add the sub-value `S` to the option `'shortmess'`:

   ```
   :set shortmess+=S
   ```

4. To remove the sub-value `S` to the option `'shortmess'`:

   ```
   set shortmess-=S
   ```

5. To set the default value to the option `'shortmess'`:

   ```
   :set shortmess&
   ```

6. We can search in Vim's help to look at the short name of the option `'clipboard'`:

   ```
   :help 'clipboard'
   ```

   Don't forget the single quotes `'` to specifically search for a Vim option.
   We can see in the help file that the short name of this option is `'cb'`.

Solution 01 – The Power of Vim Search

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

Solution 02 – Search Highlighting

1. To search for the word "vim":

   ```
   /vim<enter>
   ```

2. To turn off the search highlighting:

```
:set nohlsearch
```

3. To turn on the search highlighting while typing a search pattern:

```
:set incsearch
```

4. To toggle the two options from step 2 and 3:

```
:set incsearch! hlsearch!
```

We turned off `'hlsearch'` in step 2, so it's now turned on.
We turned on `'incsearch'` in step 3, so it's now turned off.

Solution 03 – Count in Action

1. To insert four times the word Vim and move the cursor at the beginning of the line, hit the following from NORMAL mode:

```
4ivim <esc>0
```

The result:

```
vim vim vim vim
```

2. To select the first three words of the line, hit the following from NORMAL mode:

```
v3aw
```

The result:

```
vim vim vim vim
```

To cancel the selection, simply hit `<esc>`.

3. To yank the current line and copy it three times below, hit the following in NORMAL mode:

```
yy3p
```

The result:

```
vim vim vim vim
vim vim vim vim
vim vim vim vim
vim vim vim vim
```

4. To create four new lines, inserting a star `*` at the beginning of each odd ones:

```
2o*<enter><esc>
```

We insert two times a line with a star `*` and a line without, so four lines in total.

Solution 04 – Vim Messages

1. To display the output of the shell command `file` with the current file as argument, and without saving this output in the message history:

   ```
   !file %
   ```

   Another solution:

   ```
   :echo system('file ' .. expand('%'))
   ```

   We'll see how to run shell commands in Vim in rank 09, and the function `expand()` is explained in rank 13.
2. The output of the Ex command from step 1 wasn't saved in the message history, but we can still hit `g<` in NORMAL mode to display the last message.
3. To search for the word "emacs" in the current file:

   ```
   /emacs<enter>
   ```

4. To clear all the messages from the message history:

   ```
   :mes clear
   ```

5. We can still display the last error message even if we cleared the message history in step 4:

   ```
   :echo v:errmsg
   ```

6. To write an entry in the message history with the full path of the current file, its `'filetype'`, the current line number, and the word under the cursor:

   ```
   :echom expand('%:p') '|' &filetype '|' line('.') '|'
   ↪  expand('<cword>')
   ```

Solution 05 – Setting Vim Options

1. We can verify that an option can have a local value thanks, as always, to Vim's help:

   ```
   :help 'shortmess'
   ```

   This specific option is only global.
   We'll look at the possible scopes for an option in rank 07.
2. To turn off the option `'expandtab'` locally:

   ```
   :setlocal noexpandtab
   ```

3. To set the option `'listchars'` easily:

   ```
   :let &l:listchars = 'tab:> ,trail:±'
   ```

We can use a digraph to insert the character `±`; simply hit the following INSERT mode keystroke:

```
CTRL-k +-
```

We're covering digraphs in rank 13.

4. In Vim (not in Neovim), we can set the following two options to have a cursor block in NORMAL mode, and a cursor line in INSERT mode:

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

For this functionality to work, your terminal needs to support it.
You can configure your cursor way further in Neovim; see `:help 'guicursor'`.